ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica

Design and Implementation of the μMPS2 Educational Emulator

Tesi di Laurea in Progetto di Sistemi Virtuali

Relatore:
Prof. Renzo Davoli

Presentata da:
Tomislav Jonjic

Sessione II
Anno Accademico 2011-2012

Design and Implementation of the μMPS2 Educational Emulator

Tomislav Jonjic

Abstract

An arguably critical part of any computer science curriculum consists of experience in designing non-trivial software systems. A first course in operating systems provides adequate ground for this endeavor, as any reasonably featured operating system is an intrinsically complex program. An operating system builds layers of abstraction upon the bare machine interface; understanding—or better yet devising—such a system encourages one to acquire a solid command of software engineering principles and heuristics. Modern machine architectures, it is argued in this thesis, are prohibitively complex and thus unsuitable as foundations for educational or experimental proof-of-concept operating systems. A preferable alternative, then, is provided by emulators of reasonably realistic but at the same time pedagogically sound hardware platforms. This thesis builds upon one such system, namely μMPS [1]—a tool primarily devised as an aid in operating systems and beginner-level computer architecture university courses. μMPS features an accessible architecture, centered around a MIPS R3000 processor, that includes a rich set of easily programmable devices.

The first major revision of μMPS is the result of the present thesis. Among the prominent features of this new version, dubbed μMPS2 [2], are multiprocessor support and a more sophisticated and easier to use user interface. The primary goal of the architecture revision was to make the system reflect reasonably well modern commodity hardware platforms, for which the clear trend over the past several years has been a shift towards multi-core designs.

After an overview of the machine architecture and the emulator environment, a relatively thorough exposition of the emulator internals is given. Considerable care was taken to ensure that the code base is accessible enough to foster further experimentation, for which possible directions are given in the last chapter.

Progettazione e implementazione dell'emulatore didattico μMPS2

Tomislav Jonjic

Sommario

Una componente importante di qualsiasi curriculum universitario di informatica è indubbiamente l'esperienza nella progettazione di sistemi software di non banale completessità. Un corso introduttivo di sistemi operativi rappresenta un contesto opportuno per questo scopo, visto che un sistema operativo è in generale un programma intrinsecamente complesso. Un sistema operativo va visto come una serie di livelli di astrazione, a partire da quello basato sull'interfaccia esposta dalla macchina stessa. La comprensione, ed a maggior ragione la progettazione, di un tale sistema non può prescindere da un'adeguata acquisizione di conoscenze dall'ambito dell'ingengeria del software. Le piattaforme hardware moderne, si sostiene in questa tesi, sono eccessivamente complesse, e di conseguenza non adatte come basi per sistemi operativi didattici oppure quelli altamente sperimentali. Un'alternativa migliore consiste nell'impiego di emulatori di piattaforme hardware ragionevolmente realistiche ma allo stesso tempo di concezione espressamente didattica. Questa tesi è basata su un tale sistema, chiamato μMPS [1]—uno strumento ideato principalmente per l'impiego sia nei corsi universitari di sistemi operativi che quelli introduttivi di architettura degli elaboratori. μMPS vanta di un'archiettura pedagogicamente accessibile, in base alla quale si trova un processore MIPS R3000 e che include un'ampia serie di dispositivi facilmente programmabili.

Il primo considerevole aggiornamento di μMPS è il risultato della presente tesi. Tra le caratteristiche notevoli della nuova versione, denominata μMPS2 [2], troviamo il supporto multiprocessore e un'interfaccia utente più usabile e sofisticata della precedente. L'obbiettivo principale della revisione dell'architettura è stato quello di rispecchiare meglio con μMPS2 le comuni piattaforme hardware moderne, per le quali si è vista una chiara e crescente tendenza verso le architetture multi-core.

Dopo una panoramica dell'architettura di μMPS2 e dell'interfaccia utente, segue un'esposizione relativamente approfondita dell'implementazione dell'emulatore. Durante lo sviluppo di questo si è posta particolare attenzione alla mantenibilità e semplicità del codice, con lo scopo di incoraggiare futuri sviluppi; alcune direzioni possibili per questi sono date nell'ultimo capitolo.

Contents

List of Figures

List of Tables

Listings

Notational Conventions

Several notational and typographical conventions are employed throughout the text:

- A bold typewriter-like typeface is used for machine registers (including processor registers *and* device registers) and register fields, as in **`Status`**;
- **`R`**.**`F`** denotes the field **`F`** of register **`R`**;
- Processor exception mnemonics are typeset in italic, as in *AdES*;
- A typewriter-like typeface is used for instructions, file names, identifiers, and code fragments, as in `jalr`.

Chapter 1

Introduction

1.1 Background

We begin this exposition with a brief account of the motivations behind the original work we build upon in this thesis. Rather than for its own interest or value, we choose to do so mainly because it will help elucidate many design choices described later on in this document.

Most computer science curricula have for a long time included a course in operating systems [3, 4]. This is rather fortunate, since many other areas of computer science find application in operating systems and, conversely, many notions studied within operating systems research are likely to be useful in other subfields of computer science. Operating system kernels of even moderate sophistication will undoubtedly make use of a variety of well known algorithms and data structures. An undergraduate course in operating systems also represents a very natural setting for a first introduction to concurrency. Finally, the course offers an outstanding opportunity for the student to review the *engineering aspects* of computer science. An operating system is an inherently complex computer program, whose design and implementation ultimately depends on the student's ability to apply sound techniques of abstraction to manage an otherwise overwhelming complexity. By participating in the design and implementation process, the student can acquire experience in structuring large software systems in general.

For reasons we have outlined above, a course in operating systems should, rather than adopting a purely descriptive approach, ideally include in its program the study and possibly an implementation of a complete operating system. Such an approach to teaching operating systems is by no means new, as evidenced by several influential textbooks devised, at least in part, to support it. As notable examples, we mention Lions' book on the 6th edition UNIX system [5], Bach's book on the internals of UNIX System V Release 2 [6], and Tanenbaum's book on MINIX [7]. Beside having had considerable influence on future operating systems course curriculums, it is not unreasonable to argue that the aforementioned works had an impact on the industry as well. All of the mentioned books describe existing, complete, operating systems. At the other extreme, the instructor may choose to assign the task of designing and

implementing the operating system to students (with proper guidance, of course). While both approaches unquestionably have their own merits, the latter arguably results in educationally more rewarding experiences [8, 9].

An instructor that chooses to assign the task of developing an operating system to students is unavoidably posed with a problem: which machine architecture should the course target? One obvious answer would be to simply use any variant of modern architectures currently in use—after all, shouldn't one focus on technologies that are relevant today and thus likely to remain so for the foreseeable future? Unfortunately, modern hardware is overwhelmingly complex—coming to terms with the quirks and complexities of today's hardware would simply take a disproportionate amount of time for a typical one-semester course. μMPS is an educational computer system architecture and emulator that was designed specifically to be pedagogically sound: it provides a streamlined version of most features typically found on modern computer systems. We argue in this thesis that μMPS is an ideal fit for the above-mentioned use case.

1.2 The μMPS Project

The goal of the μMPS project, as was already noted, has been to develop a computer system architecture and supporting software— that is, the "courseware", of which the machine emulator constitutes the most important part—especially tailored to computer science education. This reflects in two often contrasting requirements: on one hand, the architecture should be representative of real ones (that is, reasonably realistic), and on the other hand it should be considerably simpler than those.

The original version of the architecture and accompanying courseware, called *MPS* [10], were authored at the University of Bologna by Mauro Morsiani, under the supervision of Renzo Davoli. The system was centered around a single MIPS R3000 processor, a member of the MIPS architecture line, which was at the time already becoming well established in computer science education as a prime example of an elegant, clean, instruction set architecture. The system specification also detailed a rich set of peripheral devices and a system bus along with their supporting controllers. Later revisions of the architecture and courseware, including the latest μMPS2, remained identical in spirit to the original version. (We present a brief introduction to the architecture in Chapter 2; for a detailed and authoritative description we refer the reader to [11].)

μMPS [1] was a slight evolution of MPS, motivated by the experience and feedback from using MPS in undergraduate operating systems courses taught by Renzo Davoli at the University of Bologna and Michael Goldweber at Xavier University. On a hardware level, this version introduced a streamlined and more orthodox virtual memory subsystem. Interestingly, the new virtual memory subsystem was implemented almost entirely using ROM level abstractions and required only minor modifications to the processor architecture. On the emulator level, μMPS featured a more novice-friendly graphical user interface.

The μMPS architecture is, as of writing, in its second major revision, labeled μMPS2, and implemented by the 2.x series of the emulator. Multiprocessor support is the single most important feature of μMPS2. With this addition, μMPS remains comparable in feature with the current generation of consumer-class computer systems.

1.3 Applications of μMPS

The original inspiration for the μMPS project, and perhaps still the driving motivation and most important use case, is the one we have already given above: to provide a didactically sound hardware platform for use by educational—whether academic or hobby—operating system projects. We see μMPS as ideally suited for this use case because it represents an acceptable compromise between realism and simplicity. Indeed, the current revision of the architecture does not deviate appreciably from that of a contemporary workstation or small server class system. At the same time, the various hardware subsystems in μMPS exclude the gratuitous complexity present in virtually all real-world hardware that would unnecessarily hinder the learning process.

μMPS can also be a valuable aid in a first introduction to MIPS assembly language programming. This topic is often selected as a small but important part of an introductory course in computer architecture, since it can help explicate many aspects of the hardware/software interface. An assembly level MIPS simulator such as SPIM [12] or MARS [13] is typically used to execute programs written in MIPS assembly. Unlike μMPS or other *emulators*, these programs do not interpret actual machine code; instead, they interpret MIPS assembly language programs directly. Consequently, using μMPS requires slightly more effort compared to an assembly simulator, since a development toolchain (an assembler and linker at minimum) is needed to prepare programs for the machine. The features provided by μMPS, such as its support for many peripheral devices or the emulator's debugging features may well be worth the additional effort.

Finally, we believe that μMPS provides a good basis for experimentation in computer system emulation, especially in the area of educational applications. The primary reason for this is the relative simplicity of the μMPS system *and* emulator, compared to prominent full system emulators, such as QEMU [14]. Since these emulators are expected to efficiently execute real-world operating systems and realistic workloads, they put first and foremost an emphasis on performance, at the cost of code complexity. As such, these highly optimized systems are inevitably both less suited for experimentation and less amenable to extensions. We give some possible directions for future extensions in Chapter 5.

1.4 Contributions of This Work

Only several years ago, consumer desktop systems were virtually without exception uniprocessor systems. Multiprocessor architectures were economically viable only for server systems and, to a lesser extent, specialized high-performance workstations. In the meantime, however, the industry has has shifted focus to multiprocessor designs for high-end and commodity systems alike. This was primarily a result of ever more diminishing returns from instruction-level parallelism and of power issues. Multi-core machines are now ubiquitous on desktop or even mobile hardware and this trend is most certainly going to continue. As a result, general purpose operating systems designs—whether educational or not—that target only uniprocessor systems are at the very least considered obsolete. The single processor design of μMPS was in this regard a serious limitation.

This thesis resulted from the attempt to bring μMPS to the era of thread-level parallelism. μMPS2 includes relatively sophisticated multiprocessor support, modeled after existing hardware but appreciably streamlined in comparison.

Extending the μMPS emulator to support the μMPS2 architecture created an opportunity to reconsider various design choices behind the emulator. An important aspect—certainly the most visible to end users—in which the μMPS2 emulator differs from its predecessors is the user interface, which has been redesigned to better fit modern user interface standards and the expectations of today's users.

1.5 Organization of this Document

This chapter introduced μMPS, its scope, and its use. In the remainder of this work, we present an overview of the μMPS2 architecture and emulator (referring the reader to the official documentation for full details) as well as information on the emulator internals.

Chapter 2 outlines the μMPS2 architecture. Although finer details of the architecture are not given (such as device controller programming information or details on machine registers), it contains enough background material to allow the reader unfamiliar with the architecture to follow later chapters. In addition, Appendix A describes in detail the changes in the μMPS2 revision of the architecture.

Chapter 3 gives a user's perspective of the μMPS2 emulator. It describes the most important parts of the user interface and the means by which emulated machines can be created and executed.

Chapter 4 dwells on the internals of the μMPS2 emulator. The material will especially be of interest to the reader who is required to understand the emulator code (that is, the prospective maintainer or contributor). Both the emulation backend and the front-end (user interface) components are described in fair detail.

Finally, in Chapter 5 we attempt to give an objective view on the overall success of the μMPS project thus far. We also list possible directions for future work. In particular, an ongoing experimental project is described that aims to mitigate the

multiplicative slowdown due to emulation of multiprocessor machines, by exploiting thread-level parallelism at the host level (at the expense of deterministic execution).

Chapter 2

An Overview of the μMPS2 Architecture

2.1 Introduction

The μMPS hardware platform is, by design, considerably easier to program than ones typically found in actual computer systems today. This design goal is reflected both in the choice of the base instruction set architecture and (especially) the device controller interfaces that comprise the system. In this chapter, we briefly describe these design choices and give a short overview of the μMPS2 architecture, with the intent to present just enough details to allow one to comfortably follow the remainder of this work.

2.2 Overall System Structure

μMPS2 includes features commonly found in a modern server or workstation class multiprocessor machine, albeit in simplified form. In a nutshell, the μMPS2 computer system is composed of:

- Up to sixteen MIPS R3000-style processors (μMPS2-specific parts of the instruction set architecture are described in Section 2.3).
- Device controllers for five device types (terminals, disks, tape readers, printers, and network adapters).
- Various support hardware, including a programmable multiprocessor interrupt controller.
- A system bus connecting the above units. The bus controller integrates a system clock and an interval timer. These devices are interfaced through a memory mapped register interface, which also includes registers that provide critical system information.

2.3 Processor Architecture

The μMPS2 processor architecture is based on the one implemented by MIPS Computer System's R2000 and R3000 models, the earliest members of the MIPS line of processors. This architecture, labeled MIPS-I, grew out of the research project of the same name at Stanford University, led by John L. Hennessy [15]. The key insight behind the project was that an instruction set composed of relatively simple operations was amenable to an efficient implementation using the technique of *pipelining*, the first of many micro-architectural techniques that were used by subsequent designs to exploit *instruction level parallelism*. The MIPS design was commercialized in 1986 by MIPS Computer Systems Inc., in the form of the R2000 processor.

The choice of instruction set architecture (ISA) was motivated by the MIPS architecture's virtue of being both didactically sound and well supported by existing compilers and tools. The MIPS-I instruction set is arguably the most elegant 32-bit ISA among those server and workstation-class architectures that have seen widespread adoption in the industry. The best evidence of this is its use in computer science education; as just one concrete example, we mention Hennessy and Patterson's influential introductory textbook on computer architecture [12].

2.3.1 μMPS2-Specific ISA Features

The instruction set implemented by the μMPS2 CPU is, from a user-level programming perspective, a strict superset[1] of the MIPS-I ISA, as implemented by the R2000 and R3000 processors. This is of fundamental importance for the μMPS project, since this level of (backward) compatibility means that μMPS can be automatically supported by existing MIPS compilers.

It is in the system control coprocessor (known as CP0 across MIPS architecture revisions) inteface—relevant from a system programming perspective—that μMPS2 slightly departs from the R3000 processor. Such incompatibilities are irrelevant for compiled code and most (if not all) higher-level software in general, since the CPU control interfaces are never targeted by compiled code. Indeed, prior to the MIPS32/64 revision of the MIPS architecture, the system control coprocessor's interface was implementation dependent; in this sense, the μMPS2 CPU is a strictly conforming implementation of the MIPS-I ISA.

Memory Management Support

Like the R3000 family of processors, the μMPS2 processor integrates support for a paged memory management scheme; the key hardware unit behind this support is the on-chip translation lookaside buffer (TLB), which in μMPS2 is of user-configurable size. On all MIPS architectures, the TLB is entirely software-managed;

[1] μMPS2 does not include floating-point support, an optional part of the MIPS-I ISA defined in the coprocessor 1 encoding space.

the notion of *page table* in particular is not defined by the hardware architecture at all.

Program (or "virtual") addresses in the R3000 and any later MIPS CPU are always subject to a form of translation—a program address is never equal to the effective physical address output by the CPU. μMPS2 introduces two modes of operation for the processor's memory management subsystem:

- a *physical memory* mode for which address translation via the TLB is not employed and program addresses correspond to physical ones in a straightforward manner;
- a *virtual memory* mode, in which address translation is used for all addresses apart from those in the range reserved for memory mapped I/O and ROM code.

As described in Section 2.5.2, the standard firmware supplied with μMPS2, building on the low-level virtual memory support, introduces a hybrid segmented-paged scheme which is for most purposes more convenient from an operating system's programmer perspective.

Cache Control

Most architectural components of the R3000 used specifically for cache management are not included in μMPS2. This is for instance the case with all the fields in the R3000 CP0 Status registers that are used for cache control and diagnostics. From a practical point of view, each μMPS2 processor can be thought of as fully cache-coherent.

Integrated Interval Timer

Like recent MIPS processors, each μMPS2 processor includes an on-processor programmable interval timer. Readers familiar with the MIPS32/64 architecture revisions should be wary that, while the timer performs the same function as the one provided by the MIPS32/64 **`Count`** and **`Compare`** registers, it exposes a different programming interface.

Instruction Set Extensions

In addition to features added via the implementation-specific CP0 registers, μMPS2 augments the core MIPS-I instruction set with some new instructions:

- The `wait` instruction, borrowed from newer MIPS ISA revisions, is used to pause the CPU until an external event occurs. This instruction is analogous to the *HLT* instruction in the x86 architecture, for example.

- The `cas` instruction is a version of the well known *compare-and-set* (or *compare-and-swap*) atomic instruction, adopted by several contemporary architectures, among which are SPARC v9 and x86-64.[2]

2.4 Devices and Interrupt Management

μMPS2 supports device controllers for five different types of peripheral devices:

- **Disk devices**
  μMPS2 disk devices are classic DMA-capable hard disk drives of configurable geometry.
- **Tape devices**
  Tape drives in μMPS2 are read-only devices. Like disks, tape devices support DMA.
- **Network devices**
  μMPS2 supports DMA-capable Ethernet adapters.
- **Printer devices**
  Printers in μMPS2 are text-only output peripherals, attached on a 8-bit parallel interface.
- **Terminal devices**
  These devices, used for text input and display, are a simplified version of the classic serial text terminal. Terminal devices are physically divided into two devices: a transmitter and a receiver.

Up to eight instances of each device type can be included in any μMPS2 machine, each one supported by the corresponding device controller. Device controllers are programmed via memory mapped hardware registers. The register-level interface is to a large extent uniform across device types.

2.4.1 Interrupts in μMPS2

All device controllers in μMPS2 support an interrupt-driven programming model: a device operation is requested by setting appropriate hardware registers; upon completion, an interrupt is generated and device registers are updated accordingly; the

[2] MIPS architectures levels starting from MIPS-II include a pair of instructions called *load-linked* and *store-conditional* (LL/SC) instead of CAS for the purpose of building synchronization primitives. Emulating LL/SC efficiently proves to be considerably more difficult, however. Since compatibility with MIPS-II and later instruction sets was not a requirement for μMPS2, this was an important consideration in the selection of an atomic read-modify-write primitive for μMPS2. As shown in [16], the expressive power of compare-and-set matches that of LL/SC. Furthermore, CAS and LL/SC are *universal primitives*, which means, in simple terms, that they can be used to implement a *non-blocking implementation* of any other atomic read-modify-write sequence. The same is not true of some other well known atomic read-modify-write primitives, such as *test-and-set* or *fetch-and-add*.

interrupt is acknowledged by issuing a new command to the device or by an explicit *acknowledge* command. The hardware does not provide any support for interrupt prioritization, but such schemes can be easily implemented in software.

2.4.2 Interrupt Line Assignment and Source Resolution

The various interrupt sources in μMPS2 are statically assigned to CPU interrupt lines (the lines represented by the **`IP`** field of the **`Cause`** register). All interrupts originating from disk controllers, for example, are assigned to interrupt line 3. While slightly inflexible, the advantage of this scheme is that it does not require configuration at the hardware level nor complex probing mechanisms from the operating system. Devices of the same class (i.e. devices sharing an interrupt line) are distinguished by a *device number*.

Because multiple interrupt sources can in general be assigned to the same line, a discovery mechanism is needed to determine which (if any) interrupts are pending at any given moment. The interrupt controller's memory mapped register interface includes a data structure, called the *interrupting devices bitmap*, which at any time indicates the interrupt state of all active sources. An analogous structure, called the *installed devices bitmap*, indicates which of the interrupt sources assigned to devices are active.

2.4.3 Interrupt Management in Multiprocessor Systems

The hardware parallelism of a multiprocessor system can be exploited by the operating system to improve interrupt servicing. Using multiple CPUs to service interrupts can potentially lead both to reduced interrupt latency (the elapsed time between the generation of the interrupt and the invocation of the respective handler routine) and increased throughput.

μMPS2 allows for fine-grained control over the distribution of interrupts to available processors. The operating system can specify the manner in which interrupts from individual sources are distributed to target CPUs by appropriately initializing a structure called the *interrupt routing table* (**`IRT`**). This structure consists of a set of memory mapped registers, each of which specifies interrupt routing parameters for a single interrupt source (e.g. *the second disk device*).

2.5 Higher Level Abstractions Through Firmware

In order to function properly, a μMPS2 machine needs to be supplied with some basic ROM code. In particular, two architecturally-defined exception entry points (one for TLB related exceptions and another for all other exceptions) lie in the ROM code region. The standard μMPS2 ROM code provides some exception processing and TLB-handling services that are likely to be more convenient to use for most OS authors than the plain hardware facilities.

The interfaces described in this section, unlike those covered earlier, are not strictly part of the μMPS2 "architecture"; instead, they are higher level abstractions devised to make μMPS2 more approachable by beginners. These interfaces are implemented via the standard ROM code (i.e. the μMPS2 "firmware") and their use is entirely optional; indeed, it is reasonable to expect that some OS authors will want to supply their own ROM code.

2.5.1 Higher-Level Exception-Handling Interface

The MIPS architecture provides minimal support for exception handling. After an exception is triggered, only information that is strictly necessary to discover its cause is placed in control coprocessor registers. In particular, the processor does not save any registers to memory on an exception. Similarly, control is transferred to one of the two predefined exception vectors, despite the variety of exception types defined by the architecture (interrupts, TLB-related exceptions, system calls, program errors, etc.).

The above is in stark contrast to the elaborate exception-handling support provided by architectures such as the x86. The μMPS2 ROM code in effect emulates some of the exception-processing support offered by CISC-like architectures. Most importantly, the ROM exception handler supports automatic saving and restoring of whole processor states: on any exception, the CPU state is saved in a previously agreed upon location for the type of exception in question; similarly, control is transferred to the OS by loading a CPU state from an OS-initialized location. (In the μMPS2 documentation, these locations are referred to as the *new* and *old processor state areas*.)

2.5.2 Higher-Level Virtual Memory Interface

The basic hardware support for virtual memory in the MIPS architecture is very rudimentary. The hardware has no notion of a page table, and consequently the burden of managing the TLB (and in particular that of TLB refill[3]) falls entirely to the OS. This is very much different from the x86-like paging model, typically described in textbooks on operating systems. Support for a rather traditional segmented-paged scheme has been thus added to μMPS, and it consists of:

- definitions of *page table* and *segment table* formats;
- extended page table-related exception types.

Like the extended exception handling support described above, this abstraction is supported by the standard execution ROM code. On (architecturally defined) *TLB*

[3]Once again, the delegation of the TLB refill mechanism to software is typical of RISC architectures, and MIPS in particular. Since TLB refill exceptions are relatively frequent in a system running an operating system that supports virtual memory, the hardware *does* offer some support in this case, however. In all MIPS CPUs, TLB refill exceptions are for performance reasons given a separate entry point, to avoid the cost of a dispatch to an exception handler subroutine.

refill exceptions, the ROM TLB exception handler is invoked. The task of this handler is that of inserting (if possible at all) the missing translation entry in the TLB or, if the entry cannot be found in the page table, "pass up" the exception to the OS.

2.6 Further References

The μMPS2 machine architecture that was outlined in this chapter is defined in [11]. The reader familiar with μMPS will find in Appendix A a description of all the changes from the earlier version of the architecture.

The MIPS I instruction set architecture—on which μMPS and μMPS2 are based—is detailed in [17], among other places.

Chapter 3

The μMPS2 Emulator: A User's View

3.1 Introduction

Using a hardware emulator, as opposed to a physical machine, for the task of developing a program for an unhosted environment (e.g. an operating system) comes with enormous advantages. As an example, the tedious task of rebooting a physical machine in order to reload a modified operating system reduces to a simple recompilation of the program followed by a suitable "reload" command to the emulator. Likewise, an emulator is usually a far more convenient debugging environment than a physical machine. Debugging capabilities can either be supported by the emulator itself, or by way of an interface to an external debugger.

This chapter describes the μMPS2 user interface environment and the means which the μMPS2 emulator puts at programmers disposal for debugging guest code. We also include basic information on the *host* side of the development environment—that is, the process of preparing programs for execution under the emulator.

3.2 User Interface Organization

The machine monitoring and debugging features present in μMPS2 result in a large amount of information at the presentation level. To avoid excessive clutter, the μMPS2 user interface (UI) is arranged into several top-level windows:

- The *main window*, shown in figure 3.1, is the central application window in μMPS2 and the only one visible by default. In its four tabbed sections, it displays machine status information and contains most of the UI elements related to machine execution control, including debugger-related variables. We describe each tabbed section in more detail at appropriate points below.

- Information about each processor is shown in the aptly named *processor window*. Displayed data includes processor registers, translation lookahead buffer

(TLB) entries, and a disassembly of the currently executed section of the program.

- For each terminal device in µMPS2, there is a top-level *terminal window* that acts as its front-end.

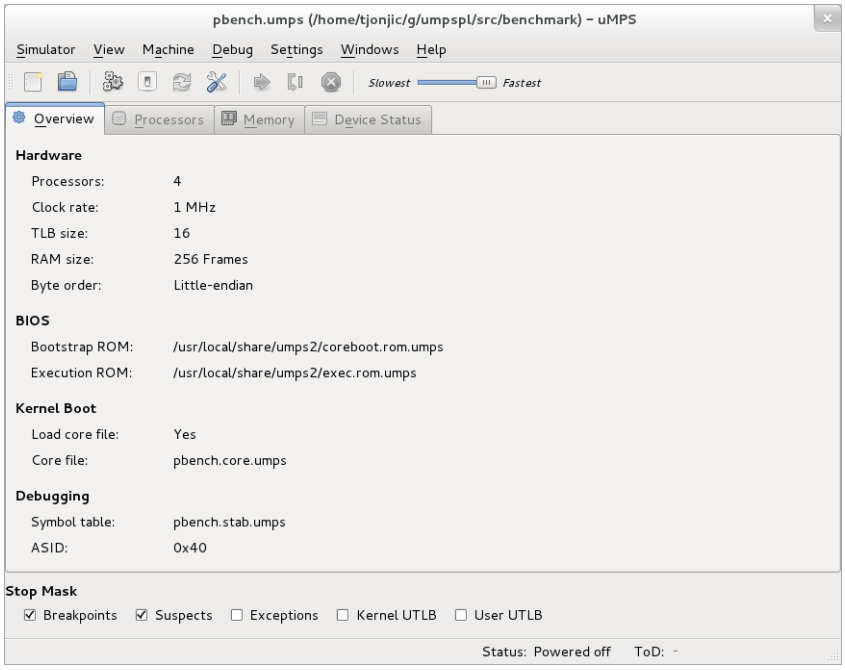


Figure 3.1: *The main window*. The central content is divided into four tabbed sections: *overview*, *processors*, *memory*, and *device status*.

Aside from the top-level windows, several transient windows (i.e. *dialogs*) are used for such tasks as editing machine configurations and breakpoint insertion.

The µMPS2 user interface is fairly flexible and configurable. For instance, window layout (placement and dimension) is persisted across sessions (at least for desktop environments that support this feature). Likewise, within single windows several elements—such as the presence and size of various sub-panes—can be customized.

3.3 Machine Configurations

In order to accomplish anything useful, the emulator must be provided with a *machine configuration*. A machine configuration consists of parameters which define all modifiable aspects of the emulated hardware environment. We can group the configuration parameters into the following categories:

- *Basic hardware characteristics*. These include the number of processors, their respective characteristics, and the amount of installed RAM.

- *Device information*. These parameters allow the user to specify the set of installed peripheral devices.

- *ROM images*. Two ROM image files must be provided: the *bootstrap* ROM and the *execution* ROM.

- *Debugging parameters*. These include, most importantly, the symbol table file.

- *Bootstrap settings*. Parameters pertaining the bootstrap process are specified here.

Machine configuration files in μMPS2 use a JSON-based [18] syntax. The user is not required to learn this syntax, however, since all configuration parameters can be set using the graphical user interface. Figure 3.2 shows part of the *machine configuration dialog*.

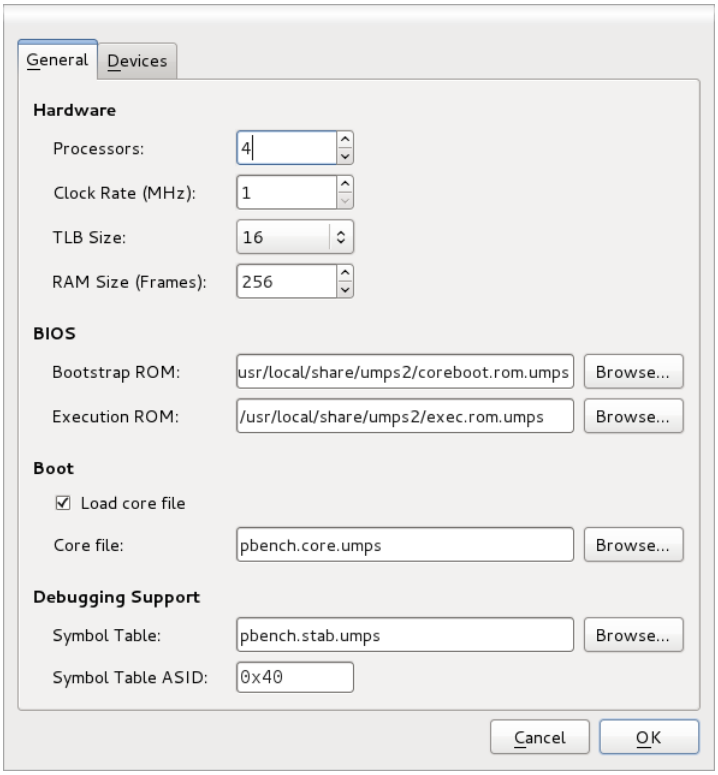


Figure 3.2: *General machine configuration parameters.*

At any given time, only a single machine may be loaded in an instance of the emulator. This is, however, not a significant limitation, since multiple application instances may naturally be launched, with each emulator instance running its own virtual machine.

3.3.1 Installed Devices and Device Files

Associated with every installed device in a µMPS2 machine is a regular file on the host's file system, called a *device file*. The precise role of the file depends on the type of device it is associated with; for simple peripherals, such as terminals and printers, it simply acts as log of the device's input and output. For non-volatile memory—that is, disks and tapes—device files act as a persistent backing store, thus allowing devices to retain data even when not powered. Like basic machine parameters, device settings can be edited using the machine configuration dialog (see figure 3.3).

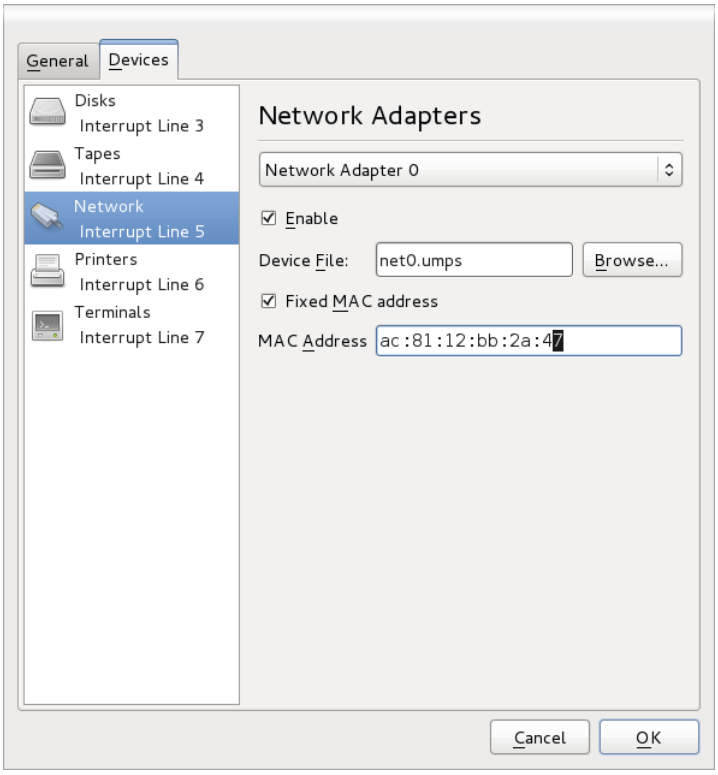


Figure 3.3: *Device settings.*

3.3.2 Byte Order in µMPS

Various computer architectures differ, among other things, in the adopted byte order scheme for native types (also commonly referred to as "endianness"). Endianness can have, in particular, important performance implications for computer emulation. If the host and emulated target endianness do not match, data needs to be converted back and forth between the two formats. To avoid this overhead, the µMPS processor always adopts the endianness of the host system.

Beside the emulator, endianness considerations are also of importance for the µMPS object file tools. In this case, the effective endianness is determined on the

basis of the input file. Thus, for example, the output file produced by the ELF to μMPS *.aout* conversion utility will use the same byte order as the (ELF) input object file.

3.4 Machine Control and Monitoring

In many respects, the μMPS2 emulator, together with its user interface, is an environment that bears similarity to that of a typical user mode program debugger. Mainstream debuggers provide an environment which grants the user fine-grained control over the execution of a program, such as the ability to temporarily suspend the execution in response to certain events (e.g. breakpoints, signals). A debugger also allows easy inspection of the program's state. In very much the same manner, the μMPS2 emulator offers a similar level of control over the execution of a virtual machine and allows inspection of the machine's state. In the rest of this section, we describe these mechanisms in fair detail.

3.4.1 Startup and Shutdown

Whenever the emulator is provided with a machine configuration, the corresponding machine may or may not be in an initialized state (that is, "powered on"). When a user command is issued to start the machine, μMPS2 validates the active machine configuration and, if possible, starts machine emulation proper. By contrast, the machine may be powered off either as a result of a user action, or as a result of an action initiated by the guest code.[1]

3.4.2 Execution Control

In the μMPS2 emulator, as in any useful debugging environment, methods are provided to request the execution of the machine to temporarily stop at specific points in the program or as a result of particular verified conditions. Once the machine has been paused, it can be inspected and debugging related variables can be modified. Execution can be resumed either by stepping through single machine instructions at a time, or by allowing the emulator to continue normal execution indefinitely—that is, until the next event of interest is verified.

The emulator can temporarily suspend execution of the guest when conditions of certain kind are verified; we will refer to those as *stop conditions*. The following stop conditions are supported by μMPS2:

- *User Requested Stops*. At any point during execution, the user can issue a stop command.
- *Breakpoints and Suspects*. These are programmer-specified stop conditions whose semantics match their direct equivalents in conventional debuggers.

[1] The emulator intercepts the hardware shutdown signal and halts the emulation loop.

- *CPU Exceptions*. The emulator can also be instructed to stop on hardware exceptions from any emulated processor.

The various types of stop conditions can be globally enabled or disabled by the user, either via a pull-down menu or via a convenient "stop mask" pane located within the main window (refer to figure 3.1). In addition, breakpoints and suspects can be toggled on an entry-by-entry basis.

Execution Model

We have thus far informally described the debugging features of the μMPS2 emulator. To avoid possible confusion, we now proceed to describe in some detail the underlying execution model. For the most part, this amounts to giving precise semantics of the various execution states that can be associated with virtual processors and the machine as a whole.

An active (i.e., powered on) machine can at any time be in one of two execution states: *running* and *stopped*. A machine can be stopped either as a result of a user-initiated action, or because an event is triggered (by the guest code) that verifies one or more stop conditions.

A processor's execution status denotes its operational condition, and its possible values form a refinement of the processor's *power states* (see Section A.1.1):

- *Halted*. This state directly corresponds to the homonymous CPU power state; the execution state reads as `halted` if and only when the power state does.

- *Running*. The processor is shown to be in a *running* state when it is in the power state of the same name and the machine's execution has not been paused.

- *Stopped*. The status implies the machine's execution has been suspended, and the processor was previously in a *running* state (conversely, a stopped machine implies that a processor status cannot read *running*). If the machine is paused as a result of a stop condition that was triggered by executing an instruction on the CPU in question, the relevant information pertaining the cause is included in the status. Thus, a possible status entry may display, for instance

  ```
  Stopped: Breakpoint(B3)
  ```

  to indicate that the machine has been paused because a breakpoint (identified by B3) had been reached.

- *Idle*. Like *halted*, this state also corresponds directly to the power state of the same name.

The execution status for all processors is displayed in the processor list pane (see figure 3.4), located in the upper half of the main window's *processors* tab.

On any stop condition, the emulator suspends execution for the machine as a whole; in other words, the *stopped* machine execution state implies that none of the

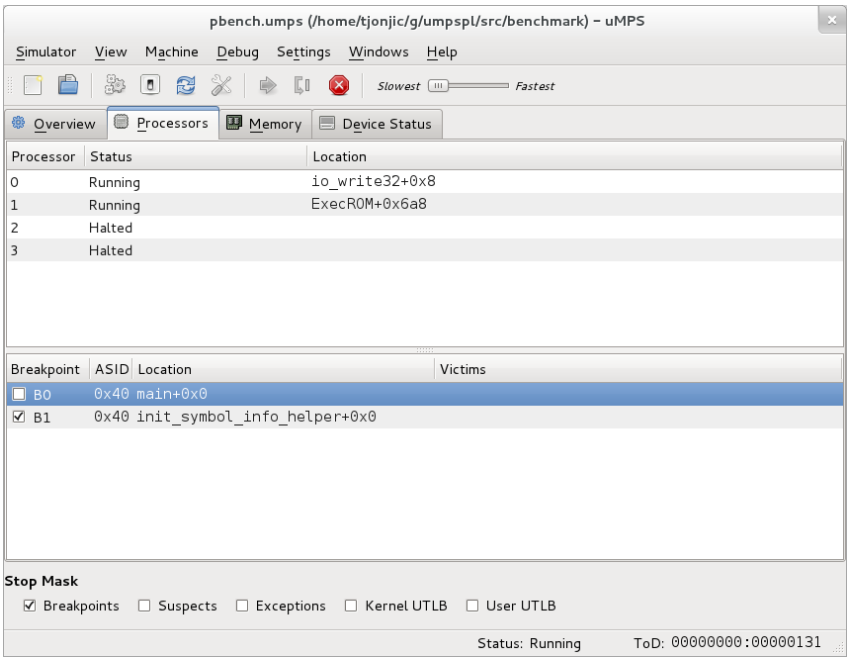


Figure 3.4: *The main window's processor tab pane.*

processors are in the *running* state.[2] As already noted above, multiple events may simultaneously cause the emulator to pause the execution of the machine.

3.4.3 Breakpoints and Suspects

As noted above, μMPS2 allows the user to define stopping points within a guest program. Readers will in all likelihood already be familiar with these notions from previous experience with traditional debuggers. Because of possible terminological friction, however, we nevertheless define the concepts precisely.

A *breakpoint* is a single, either physical or virtual, word-aligned address; the emulator is able to suspend machine execution when an instruction is fetched from the location pointed to by the address. Emulation is paused precisely *before* the instruction in question is executed.

Breakpoints can be set via the breakpoint insertion dialog (see figure 3.5) or the code view (see Section 3.4.4). Each breakpoint can at any time be enabled or disabled, a task performed via the breakpoint list pane. Additionally, inserted breakpoints can be collectively disabled, using either a menu command or the convenient stop mask pane.

A *suspect*, which is specified by a range of consecutive word-aligned addresses,

[2] Some user-mode debuggers support execution modes in which a subset of threads comprising a multithreaded program are allowed to execute while other threads are stopped. μMPS2 does *not* support an analogous execution mode that would allow only a strict subset of CPUs to proceed with execution. While this feature is arguably useful for debugging multithreaded user-mode programs, it was dismissed as being overly unrealistic (and confusing) in our context.

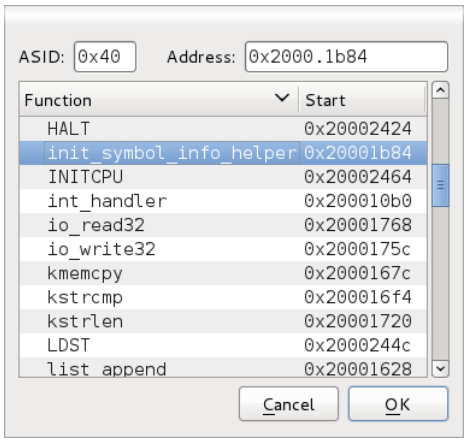


Figure 3.5: *The breakpoint insertion dialog.*

causes machine execution to be suspended on a read or write access to a memory location in the suspect's range. There are three types of suspects: *read*, *write*, and *read/write*; each of these types has expected and obvious semantics (a *read*-only suspect, for example, will cause the emulator to stop execution only on loads—but not on stores—from the memory location in question). Like breakpoints, suspects can be toggled collectively or on an entry-by-entry basis.

Suspects and breakpoints can be associated with an address space identifier (ASID). This allows (simultaneous) debugging of code running with address translation disabled (i.e. the kernel) and code running with address translation enabled (i.e. user mode processes).

3.4.4 Examining Processor State

The µMPS2 architecture supports machines with up to sixteen processors. It is often desirable during the debugging process to examine the current state of one or more processors. By "state", of course, we intend all the *architecturally visible* state. Without proper organization at the user interface level, the abundance of information could easily become overwhelming. The solution adopted by µMPS2 is a multi-window interface; each installed processor has an associated *processor window*, which can be shown or hidden at the user's preference. An example is shown in figure 3.6.

In addition to the standard menu and toolbar, the CPU window comprises several elements, some of which are optional:

- The *code view* displays a code disassembly of the currently executing function. For convenience, the user can add and remove breakpoints on the shown locations directly using the code view.

- The *register view* displays the value of general purpose registers, CP0 registers, along with some non-architectural register-like data (such as the "program counter"). The user is able to switch between hexadecimal, signed decimal,

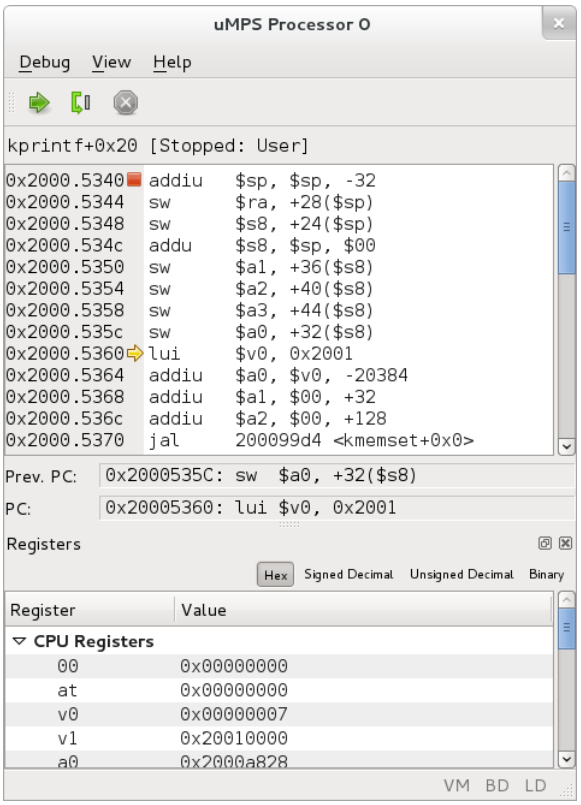


Figure 3.6: *The processor window*. The code view—which forms the core and always-visible part of the window—is shown in the upper half. Breakpoints are represented by suitable markers along the relative memory addresses; in the shown example, a breakpoint is set at the procedure's entry point. In the lower part, a dockable CPU register view is present. The TLB display, another optional dockable pane, is not present.

unsigned decimal, and binary representations of these values. Register values can be modified by the user, and input is supported in each of the aforementioned representations.

- The *TLB view* displays the contents of the translation lookaside buffer (TLB). As with CPU registers, TLB entries can be modified by the user.

3.4.5 Memory View

It is often desirable during a debugging session to inspect arbitrary memory contents. μMPS2 allows the user to define multiple address intervals, called *traced regions*, which can later be easily accessed in order to display the respective memory contents. The user interface elements of this facility are located in the *memory* tab of the application's main window, as shown in figure 3.7. The memory tab also hosts the list of memory suspect ranges.

μMPS2 currently supports two different display modes for traced data: a non-

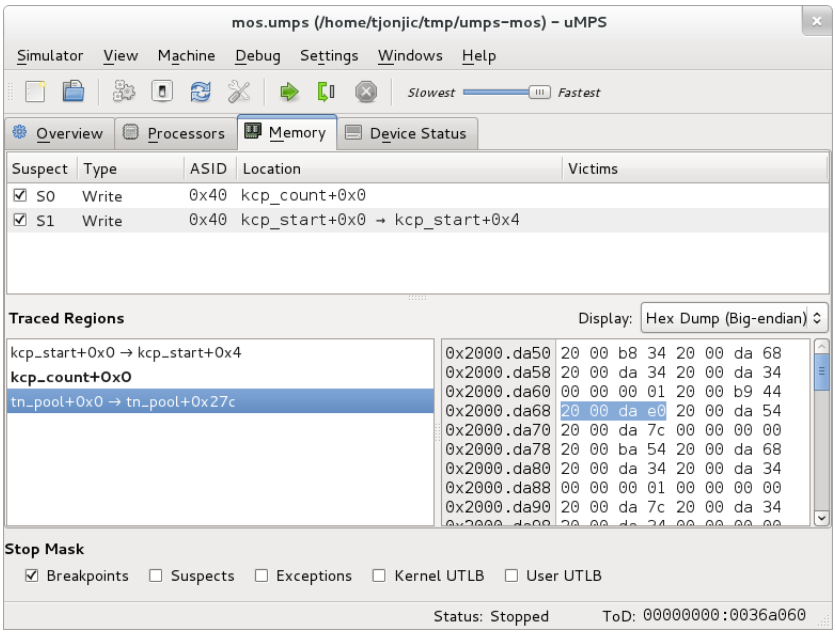


Figure 3.7: *The main window's memory tab.* The tab hosts the list of memory suspect ranges in the upper half, and the list and rendering of defined traced regions. A hexadecimal dump of the selected memory region is shown.

editable ASCII mode and an interactive "hex dump" mode that supports in-place editing of the memory contents. The former would typically be used for textual data, and the latter in any other case.

3.4.6 Device Monitoring

The last tab pane in the main window displays device status information, as shown in figure 3.8. Device information is organized into a tree, in which the top-level branches represent device types, and the leaves the respective devices. Since the various device types in µMPS2 provide a rather uniform memory-mapped register interface, a common display format was possible for all devices.

3.4.7 Terminals

µMPS2 features a single human interface device—the terminal. The *terminal window*, shown in figure 3.9, is the UI counterpart to terminal devices; in a sense, it is akin to terminal emulator packages common on modern desktop systems (parenthetically though, µMPS2 terminals do not share the complexity nor the feature set of real-world text terminals, such as VT100).

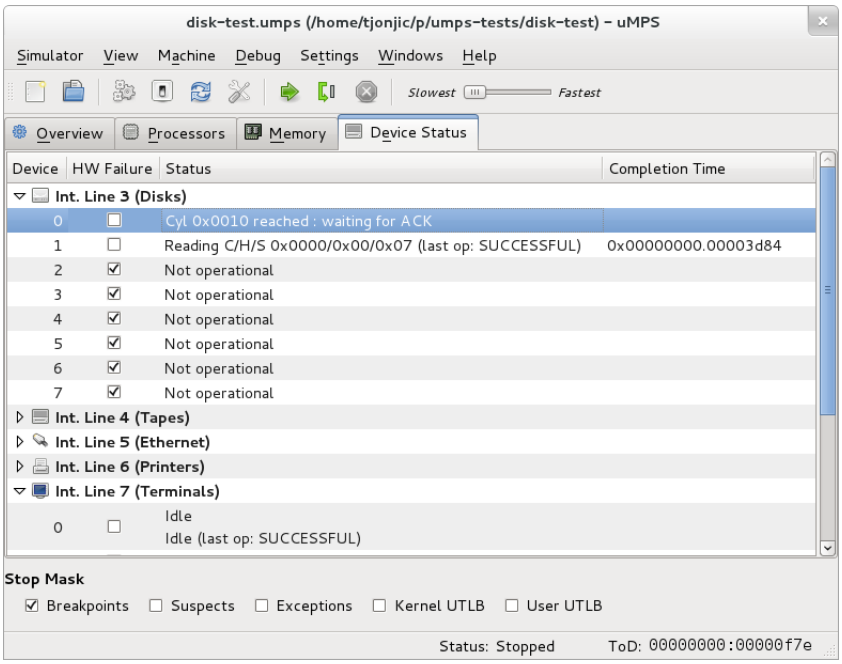


Figure 3.8: *Device status view*. Devices are grouped by type (or, equivalently, by interrupt line assignment) and each group can be expanded or collapsed according to preferences.

3.5 Programming for μMPS2

As has been already stressed, the inherent suitability of μMPS as a target platform for educational or experimental operating systems is a direct result of its streamlined machine architecture. Furthermore, because of its choice of instruction set architecture, it is readily supported by existing cross compilers. Thus, it can be claimed that development for μMPS is not substantially different from that of any other (bare-machine) platform, if only easier. There are a number of ways, however, in which μMPS further simplifies this process for the beginner. In a nutshell, it achieves that by including support for a simplified object file format, providing useful pre-built ROM images with code that implements a beginner-friendly exception handling scheme, and by supporting a much simplified boot process. This section is only meant to give a brief overview of the development process; the documentation mentioned at the end of the chapter covers the topic in detail.

3.5.1 ROM Images

The μMPS2 emulator requires the user to provide images for the *bootstrap* and *execution* ROM, which are expected to supply vital low-level code. The bootstrap and execution ROM images are mapped to locations that coincide with the bootstrap and exception vectors, respectively. At machine reset time, processor 0 begins its instruction-cycle starting at location `0x1FC0.0000`, which is contained in the ad-

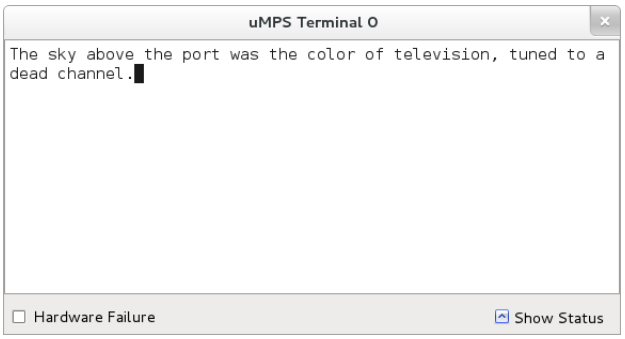


Figure 3.9: *The terminal window*. For convenience, optional device status information can be shown at the bottom.

dress range on which the bootstrap ROM is mapped to. Likewise, whenever a hardware exception is raised during normal system operation, control is transferred to location `0x0000.0080`. At this address, located in the execution ROM range, exception handling code is expected to be found.

Useful ROM images are included in the µMPS2 distribution that should suffice for most uses. The execution ROM, in particular, implements the two-phase exception handling mechanism mentioned in chapter 2 and described in detail in [11]. Advanced users can, of course, fairly easily provide alternative implementations if the need arises.

3.5.2 The Toolchain

It is hardly worth pointing out that users are expected to program for µMPS2 in a high-level language. At this time, the most viable implementation language for targeting µMPS2 is certainly C, with C++ perhaps being another plausible option.[3] Being a conforming implementation of the MIPS-I ISA, µMPS2 is supported by unmodified MIPS cross-development toolchains—that is, compilers along with the associated set of object file utilities. The toolchain that has been so far used by the project maintainers exclusively, and has been subject to extensive testing, is the GNU toolchain—formed by the GNU Compiler Collection and the GNU Binutils suite.

For completeness, we should also mention that it is conceivable that one may wish to use a standard C library for µMPS development, either for the kernel itself or (more likely) for programs hosted under it. There are several lightweight C standard library implementations that one could with some effort adopt for µMPS development. One particularly popular choice on embedded systems in general is Newlib [19], which only requires a relatively small amount of platform-specific code.

[3] Depending on personal preferences, some features of C++, such as syntactic support for single-dispatch polymorphism and generic programming, may be tempting enough for some programmers to select C++ as an implementation language for an operating systems kernel. On the downside, certain features of the language require the programmer to provide special run-time support when used in an unhosted environment. It is far from clear, then, whether the extra effort is offset by the benefits for simple projects.

3.5.3 Object File Formats

Virtually all fairly recent versions of GNU cross-development toolchains for MIPS systems use the ELF object file[4] format natively. This format, while having many virtues, requires in general somewhat complex run-time support (i.e. loaders). For this reason, μMPS includes support for a greatly simplified format that comes in two variants, called *.aout* and *.core*, based on the much older UNIX *a.out* format. This format, while not as flexible as ELF, requires much simpler run-time support and should suffice for most uses. On the other hand, the ELF format may well be preferred for more ambitious projects due to its versatility (such as its ability to embed arbitrary metadata in a structured manner). The use of ELF (or any other format) is of course entirely possible, since the emulator is in no way linked to any notion of object file format. Ready to use boot loading code that is included with μMPS2 only supports the .core variant of the .aout format, however.

3.5.4 Operating System Bootstrap

In many cases, it is advisable to enable users to start experimenting with the μMPS2 system by writing simple programs without requiring them to dwell into object file format details and boot loading code. There are two ways to achieve this in μMPS2. One option is to use the pre-built bootstrap ROM module `tapeboot.rom.umps`, which contains a simple boot loader; the program (e.g. operating system) is in this case expected to be located on the first tape device.

Another extremely convenient bootstrap option is to take advantage of the emulator's ability to pre-load the executable into memory before machine startup. The `coreboot.rom.umps` ROM module should be used in this case. In both cases the program is expected to be in the .core format.

3.6 Further References

The official μMPS2 reference [11] contains a full description of the standard ROM services, the specification of the μMPS object file formats, and programming tips. A rather in-depth guide on some MIPS-specific aspects of the GNU toolchain, aimed specifically at μMPS2 users, is [20].

[4] By the term object file, we refer in general to relocatable object files, executables, and shared libraries.

Chapter 4

μMPS2 Emulator Internals

In this chapter we describe in reasonable detail the inner workings of the μMPS2 emulator. In most cases, the discussion is limited to key design choices and the general structure of the software; the source itself remains the authoritative reference for any finer detail.

4.1 Design Principles and Overall Structure

Before dwelling into any details about the emulator's internals, we mention several core design goals that had a pervasive impact on its code base. First and foremost, the μMPS code base strives to be a convenient ground for experimentation and further extensions or improvements. Simplicity, elegance, and ease of maintenance of the resulting code, for instance, have often taken precedence over other qualities in the selection of solution alternatives for a given problem. In particular, the primary focus of the μMPS2 emulator is not uncompromising performance, as it is for emulators such as QEMU—aimed at running real-world operating systems and realistic workloads.

An important feature of the emulator is the complete independence of the core emulation code (the *back-end*) from user interface (*front-end*) code; the communication between the two takes place through a well-defined API. Thus, in spite of the fact that the sole user interface supported at the time of writing is a fully-fledged graphical one, it is entirely feasible that a command-line based interface may be supported in the future, for example.

In light of what was said above, we can group the modules comprising the μMPS2 system into several categories. First, the *emulation engine* consists of processor emulation code, various device models, as well as machine management and debugging support. A *user interface* is then provided via a series of modules that employ the API exposed by the emulation engine; it consists of a multitude of classes which display machine state, as well as user interface counterparts to debugging-related facilities. Last, we find modules belonging to auxiliary standalone programs, such as the block device creation utility, or the μMPS .aout object file creation and

inspection tools.

4.1.1 Signals, Slots, and the Observer Pattern

In the implementation of software systems of non-trivial complexity, there often arises a need for certain entities comprising the program (the *observers*) to be notified about particular events of interest related to another entity (the *observable*). What we vaguely refer to as "entities" are typically class instances, and the "events of interest" are changes in the object's visible state; that, however, is certainly not a technical requirement. Moreover, this dependency between components should preferably be established without imposing a tight coupling between them. The archetypal example of this is to be found in user interface toolkit libraries, where changes in user interface elements are usually required to trigger an appropriate reaction from parts of the program expressing some domain-specific logic.

What is desirable, then, is to have a consistent and idiomatic way of expressing the propagation of change in a program.[1] Since considerable infrastructure is needed to implement this pattern in a satisfactory manner, it is preferable to employ one of the many available libraries. While the terminology and metaphors vary across implementations, all the different APIs are centered around the basic abstract notions of *signals* and *slots*. Signals are representations of events that can be programmatically emitted. Each signal can be bound to multiple slots—specially designated functions or function-like objects that intercept the signal whenever it is emitted. Virtually all library implementations provide two main benefits over hand-maintained code:

- Boilerplate code is reduced to a minimum; for example, the infrastructure needed to maintain the list of observers for each given signal is provided by the library.
- Automatic tracking of object lifetime is provided by the library, thus avoiding the tedious task of manually unbinding signals from deallocated observers.

In μMPS2 two such implementations are used. The emulation back-end employs the highly portable and lightweight libsigc++ [22] library. Within the front-end code, the Qt framework's own signal/slot facility is leveraged, for better integration with other components of the framework.

4.1.2 Portability and the Choice of Implementation Language

The current μMPS2 project inherited its code base from μMPS, which in turn was—implementation-wise—an incremental evolution of the original MPS system. For MPS, C++ [23] had been chosen as the implementation language. It is arguable that C++, especially at the time, offered a very sound compromise between performance and high-level language features; since both of these requirements were (and still are) of uttermost importance for a program such as a computer system emulator,

[1] This idea is at the core of a programming paradigm known as *reactive programming* [21].

C++ was at the very least a justifiable choice. The μMPS2 project has retained C++ as the implementation language, and has gradually shifted from a pre-standard dialect to the C++03 standard and a slightly more liberal use of language facilities and the standard library.[2] The current versions of all widely used C++ compilers fully support this standard. The core emulator code depends on a small number of external libraries—most notably, the Boost library—all of which are portable across all major platforms.

In addition to the dependencies inherited from the emulation back-end, the resulting application further depends on the Qt libraries, on which the user interface in μMPS2 is built upon.[3]

Finally, we include a note on build automation, as it can be seen as a portability consideration. For any project of considerable size or dependency requirements, the use of a robust build automation system is likely to be vastly superior to more primitive solutions, such as hand-written Makefiles and ad-hoc shell scripts. μMPS2 currently uses the GNU build system, most notably GNU Automake and Autoconf. Despite some of its shortcomings—such as being almost inherently tied to Unix-like systems—the GNU build system (commonly referred to as the "Autotools") has so far met the project's requirements.

4.1.3 Source Tree Structure

In the rest of this chapter, various subsystems of the emulator are discussed. For ease of reference, a description of the source tree layout is given below:

`build-aux`
: Auxiliary files pertaining to the build system.

`examples`
: Self-contained example programs for μMPS2.

`m4`
: GNU M4 macros, another part of the build system infrastructure.

`src/base`
: Library of common utility classes and functions.

`src/frontends/qmps`
: GUI emulator frontend.

[2] The experience with the project has been, however, a first hand confirmation of the daunting complexity of the C++ language. This is both due to single language components in isolation (such as its complicated but nevertheless inflexible type system) and to their interoperability. The last is in many situations likely to lead to corner cases that are poorly understood by programmers or cause portability issues because of inconsistent support across different compilers. The programmer is thus often forced to adopt an austere discipline in the selection of language constructs and library facilities.

[3] The μMPS 1.x series of the emulator used the Xforms user interface toolkit.

src/frontends/qmps/data
Installed architecture-independent data files belonging to the front-end.

src/include
Installed µMPS2 headers.

src/support/bios
Assembly sources for the execution and bootstrap ROM images. For distribution packages, this directory also contains pre-built images.

src/support/crt
Start-up modules for .aout and .core executables.

src/support/ldscripts
Linker scripts for .aout and .core executables.

src/support/legacy
Various support files maintained for backward compatibility with older versions.

src/support/libumps
The libumps library.

src/umps
The emulator core.

As mentioned above, the GNU build system is currently being used to assist in configuration, build, and packaging. The source tree follows the classic recursive Automake setup for multi directory projects: in general, each directory in the project tree contains an Automake input file named 'Makefile.am'.

4.2 The Emulation Core

The µMPS2 emulator is a so called *full system emulator*: it emulates a complete computer system, including processors, memory, devices, and controllers of various types. We wish to describe here the emulator internals by grouping the code base function-wise into components. What are, then, the main tasks associated with executing a virtual machine? Very concisely, emulating a µMPS2 machine (the *guest*) consists of:

1. Executing guest code on emulated processors;
2. Emulating various devices along with their controllers;
3. Various bookkeeping tasks, such as those performed by the event handling subsystem.

The emulator's object oriented design largely reflects the organization of the hardware itself. Indeed, in most cases there is an immediate correspondence between a system component and the relative class that models its operation:

- The `Processor` class models the μMPS2 variant of a MIPS-I (R2000/R3000) processor.
- The `SystemBus` class implements the address decoding logic (i.e. mapping of addresses to memory-mapped I/O registers and physical memory locations) and the DMA support, in addition to maintaining a number of system registers.
- Each μMPS2 device type (along with its controller) is modeled by a corresponding specialization of the abstract `Device` class.
- The programmable multiprocessor interrupt controller is implemented by the `InterruptController` class.

Before we discuss any of the above in more detail, though, we look at how machine configurations and running machine instances are represented within the emulator.

4.2.1 Machine Configurations and Machine Instances

The μMPS2 emulator allows the user to manage multiple virtual machines by associating each machine with a so called *machine configuration*, which contain all user-settable machine parameters (see section 3.3). Machine configurations in μMPS2 are backed by files and use a JSON-based syntax. A sample machine configuration is shown in Listing 4.1. The syntax was designed to be as self-documenting as possible; it is easy to see at first glance, for instance, that lines 2-4 specify processor characteristics. For completeness, however, a full syntax is given in Appendix B.

Listing 4.1: A sample machine configuration.

```
1  {
2      "num-processors": 4,
3      "clock-rate": 1,
4      "tlb-size": 16,
5      "num-ram-frames": 256,
6      "bootstrap-rom": "/usr/local/share/umps2/coreboot.rom.umps",
7      "execution-rom": "/usr/local/share/umps2/exec.rom.umps",
8      "devices": {
9          "terminal0": {
10             "enabled": true,
11             "file": "term0.umps"
12         }
13     },
14     "symbol-table": {
15         "asid": 64,
16         "file": "kernel.stab.umps"
17     },
```

```
18      "boot": {
19          "core-file": "kernel.core.umps",
20          "load-core-file": true
21      }
22  }
```

The emulator currently uses a custom JSON parsing and serializing module.[4] Its API consists of a parsing component (`JsonParser`) and a JSON type hierarchy, the root being the class `JsonNode`.

Internally, machine configurations are not manipulated directly by operating on the JSON structure; rather, an abstraction is provided by the `MachineConfig` class and every other module of the emulator core and the user interface operates exclusively on instances of this class. Besides providing a more natural C++ API, the abstraction offers a more practical advantage: the implementation caches all frequently accessed configuration fields—most importantly, those that are accessed on a typical machine cycle—thus avoiding the need to retrieve them by navigating the JSON hierarchy.

Once a valid machine configuration object is available, one may finally initialize all the emulator-related structures. The top-level representation of an active μMPS2 virtual machine is an instance of the `Machine` class. For front-end code, this class provides the emulation back-end's primary interface; in particular, through its public interface the front-end may request the execution of one or more machine cycles, manage debugger-related variables, and access public virtual machine-related structures.

4.2.2 Processor Emulation

Processor emulation is undeniably the most critical part of any computer system emulator. There are in general two contrasting approaches to processor emulation. The first, considerably simpler approach, follows an *interpreter* model: the program residing on the emulated machine is executed directly (usually without any form of prior translation to an intermediate representation), by interpreting each instruction separately as it is encountered in the natural program flow. The obvious virtue of this approach is implementational simplicity: conceptually, a processor emulator of this type consists of a collection of instruction handlers and a top-level dispatch mechanism, acting on the instruction's opcode. It is, of course, unreasonable to expect that such a simple strategy would yield implementations fast enough to emulate a processor of the same class as the host's one with comparable speed. Nevertheless, for certain applications, these emulators can perform more than adequately if implemented properly. Popular emulators of older computer and video game console systems, for instance, use this kind of CPU emulation. A radically different approach to processor emulation—as anticipated by the choice of terminology—involves *dynamic recompilation* (also often referred to as *dynamic binary translation*) of the machine code

[4] `base/json.{h,cc}`

from the emulated architecture to the host's instruction set. A form of this technique is used by QEMU [14], to name one popular example. While necessary in many cases to achieve acceptable performance, this approach is considerably more ambitious and complex implementation-wise.

It is important to note that, while being a conforming software implementation of the MIPS-I instruction set, processor emulation in μMPS2 in no way mimics any of its hardware implementations. In other words, unlike system *simulators*, μMPS2 does not incorporate any realistic cycle-accurate timing model. In particular, it does not include a CPU pipeline, coherent cache, and similar models.

Each processor in μMPS2 is represented by an instance of the `Processor` class. This class naturally encapsulates all of the processor architectural state, along with some auxiliary information needed to properly support some of the MIPS architecture's idiosyncrasies, on which we elaborate in some detail below. Instruction emulation is performed in the "interpreter" style described above (see `execInstr` and the auxiliary methods: `execRegInstr`, `execBranchInstr`, and similar).

Apart from single instruction handlers, the bulk of the remainder processor emulation code concerns virtual address translation (that is, memory management unit emulation), the CPU exception mechanism, and various other auxiliary operations. Finally, the `Processor` class implements a number of public methods, which may be roughly grouped as follows:

- *Execution control:* Included here are methods used to execute a processor cycle, obtain the number of "idle" cycles (for a processor in the *idle* power state), and skip a number of cycles.

- *State querying and modification:* All of the processor's architectural state can be inspected, including TLB entries. In addition, some related non-architectural variables can be queried—notably, the "program counter".[5]

  Most of these quantities can also be modified via suitable mutator methods. Explicit modification of processor state is supported only because its practical utility to front-ends (or, rather, its users).

- *Signal control:* These methods provide a means for other system components to signal a pending interrupt or some other form of exception to the processor.

Architecturally visible pipeline effects

In spite of the relative simplicity of both the MIPS-I architecture and the emulation techniques employed in μMPS2, we find a number of subtle and non-intuitive aspects of the resulting emulation code. These are due to implementation-induced artifacts

[5]The MIPS architecture does not define any means by which a value corresponding to the intuitive notion of a *program counter* could be obtained. As with the case of delayed branches, this is due to microarchitectural restrictions. Indeed, given the pipelined nature of most (if not all) MIPS implementations, it would be far from obvious how to assign a useful and precise semantics to such a value.

in the architecture, the most famous cases being known in MIPS parlance as *delayed branches* and *load delays*.

Figure 4.1 illustrates the classic R3000-style processor pipeline, with pipeline stages being displayed for three different instructions; these are, in program order: a branch instruction; the instruction at the location immediately following the branch instruction; the instruction at the branch target. It is immediate to see from the diagram that the branch target location, by being only computed during the pipeline's ALU stage, cannot possibly be made available to the fetch stage (IF) immediately after the branch instruction itself. (Indeed, the branch target address is made available to the IF stage after a single cycle delay only because the datapath provides special handling in this case via branch-target forwarding.) Moreover, by the time the target *does* become available, the instruction immediately following the branch in memory will already have completed the early pipeline stages. The MIPS architecture mandates that the instruction immediately following the branch be always executed before the branch target instruction, thus freeing implementations from the need to resort to pipeline stalls in this case. The location immediately following the branch is known as the *branch delay slot*. To emulate this behavior properly, a three-element "stack" of program counters must be maintained instead of a single one.[6]

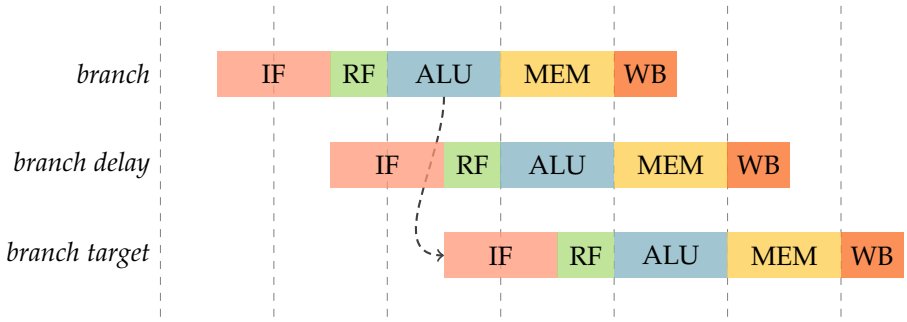


Figure 4.1: *The MIPS five-stage pipeline and delayed branches.* The classic R2000 style five-stage pipeline is shown (IF: instruction fetch; RF: register file access; ALU: arithmetic or logical operation; MEM: memory access; WB: register write back). The dashed arrow represents branch target forwarding.

An entirely analogous situation occurs with load instructions. In this case, the fetched memory value is not available to the instruction immediately following the load—called the *load delay slot*. Again, rather than forcing implementations to resort to appropriate interlocks, the architecture specification states that an attempt to use the fetched value in the load delay slot results in undefined behavior.

[6]One rather delicate case occurs in the context of exception handling. When an exception is triggered on a delay slot, the EPC register must contain the address of the *branch instruction* preceding it, in order for it to retain its usual semantics—that is, the register containing the proper *restart address*. If the program was instead restarted from the delay slot, the branch itself would never take place after an exception in the delay slot.

4.2.3 Device Emulation

Each device in a μMPS2 machine has an associated *device model*—that is, an instance of a suitable class which simulates the device's architecture-specified behavior. Naturally, one such class exists for every supported device type; for convenience, we will refer to any of these classes as a *device class*.

Every (concrete) device class is a specialization of the abstract class `Device`. Its (public) interface exposes means to read from and write to device registers, query a device status and set its operational state. Subclasses do not add to this any public interfaces of their own; hence `Device` represents the protocol devices models expose to other subsystems. Device specific emulation code is, owing to the relative simplicity of the hardware itself, rather straightforward and as such should not require further comment here.

All device controllers in μMPS2 use exclusively memory-mapped registers. Given the simplicity of the μMPS2 memory-mapped I/O space and the rather uniform format (e.g., equal size) of device registers, the mapping is performed entirely by the address decoding logic within `SystemBus`. (Had the need for greater flexibility arisen, a reasonable alternative design would be to support assignment of memory addresses to a particular device's registers at device initialization time.)

Interrupt Management

The sophisticated multiprocessor interrupt control mechanisms of μMPS2 (see Appendix A or [11]) are implemented by the `InterruptController` class. The implementation handles all memory accesses to and from locations assigned to the interrupt controller register space. The latter, we recall, consists of the *interrupt routing table* (**`IRT`**) and a set of processor-specific registers for each installed processor—notably, the *interrupting devices bitmap*. Reads and writes to the controller's registers are handled, naturally enough, by the `read` and `write` methods, respectively.

To support the multiprocessor interrupt distribution scheme, distribution (routing) parameters are maintained for each interrupt source (e.g., a specific device), and the totality of this parameters forms the **`IRT`**. These architecturally-exposed parameters alone are not sufficient for a correct implementation of interrupt distribution, however—for a rather subtle reason. When an interrupt is delivered to a target processor, the relative interrupt line is asserted, along with the respective entry in the interrupting devices bitmap. The interrupt then remains pending until acknowledged. The μMPS2 architecture, however, does not prohibit an interrupt to be acknowledged (implicitly or explicitly) via a command initiated by *another* processor. Moreover, the routing parameters for the interrupt source in question may legitimately be changed in the interim period. It is therefore necessary to maintain information about the target (if any) the last interrupt from the source was delivered to.

Along with interrupt routing, `InterruptController` contains the necessary support for the controller's processor interface registers. In particular, inter-processor interrupts are managed entirely within this class.

A complete interrupt life cycle, from the emulator's point of view, consist of the following steps:

1. Upon completion of an I/O operation, a device model issues an interrupt request by calling the interrupt controller's `startIRQ` method.
2. A target processor is determined by the interrupt controller, on the basis of the active *routing policy* and *destination* parameters. The relative interrupt line and element of the interrupting devices bitmap are asserted.
3. Upon interrupt acknowledgment, the device model ends the interrupt cycle by calling the interrupt controller's `endIRQ` method. The relative bit in the interrupting devices bitmap is deasserted and, provided there are no other pending interrupts on the line in question, so is the interrupt line. We remark again that the processor issuing the acknowledgment may be different from the interrupt's target processor.

4.2.4 Virtual Time, Machine Cycles, and Event Management

After an overview of the main hardware emulation subsystems, we shall now look at what may be considered part of the virtual machine's "execution dynamics"—that is, the way in which a machine as a whole and its various components operate with respect to "time".

Machine emulation in µMPS2 progresses in discrete time steps, corresponding to single machine cycles—or, equivalently, clock ticks. This clock is maintained by the emulator itself, and is independent of the host's time, the emulator process's wall clock time, or any other external notion of time. The operation of the machine may then be seen as a sequence of events; for our purposes, an *event* may be defined as an observable change of machine state occurring at a particular instant in time. Thus, the execution of a single instruction is (rather trivially) an event. An example of a generally more sporadic event is the completion of an I/O operation. This clock only advances when all processing scheduled for execution on the current instant (i.e., a particular machine cycle) has been completed.

Thus far, events have been mentioned without stating explicitly where those originate from, or how events are represented and managed. Clearly, we have "events" that are processed on every cycle: instruction execution for each active processor and updates of all hardware timers. The other type is constituted by *asynchronous events*, which always represent the completion of a device operation. These asynchronous events are scheduled by device models using the `SystemBus::scheduleEvent` method, which accepts an event expiration time and a callback to be invoked at event occurrence time. Internally, scheduled events are kept in a priority queue (see `event.cc`).

A pleasing consequence of the above execution model is completely deterministic execution: given identical conditions, an emulator session is *repeatable*. In other words, given the same machine configuration and device input, two emulation sessions will result in the same sequence of events.

Idle Cycles

During an emulation session, there may be intervals of inactivity; that is, intervals of idle machine cycles, during which all processors are either halted or in a low-power state (waiting for an external event to occur), and no asynchronous events are programmed for said interval. Rather than requiring front-ends to step through this idle cycles, the emulator allows front-ends to skip this potentially long intervals. At any point, the number of remaining idle cycles may be obtained via the `Machine::idleCycles` method. The machine may then be advanced by a specified number of cycles by invoking the `Machine::skip` method.

4.2.5 Debugging Support

Aside from the hardware emulation logic itself, the core emulator module also includes debugging support.

A breakpoint, suspect, or a "tracepoint" (i.e., traced region) is represented by the `Stoppoint` class. An instance of this class, which we refer to as a *stoppoint*, is essentially defined by a unique identifier and either a single memory address or a range of addresses. A collection of stoppoints is, in turn, managed by an instance of the `StoppointSet` class. Stoppoint sets support basic insertion and removal operations, lookup based on address or an address range, and index-based access with respect to insertion order.

The front-end is responsible of instantiating stoppoint sets to be associated with a machine and of supplying them to the emulator at machine initialization time. These objects are not immutable, of course, since the debugger variables they represent in general need to be modified during an emulation session.

4.2.6 High-Level View of the Emulation API

Anticipating the next section on the μMPS2 user interface implementation, it is worth to outline the public emulation API and its typical usage pattern by front-end code.

Interaction between any (hypothetical) front-end, whether GUI or command line interface-based, and the emulator core will in general involve the following steps:

1. Creating an instance of `Machine` (see `machine.h`) from a valid machine configuration object. Objects that manage the breakpoint, suspect, and tracepoint collections also need to be supplied to the class constructor.

2. Executing machine cycles; essentially, this simply consists of invoking the `Machine::step` method. Since machine execution is an inherently compute bound task, however, its integration with the remainder of the application's processing (user input processing, for instance) is a considerable challenge.

3. Issuing control and debugging commands: managing the "stop mask" (see `Machine::setStopMask` and `Machine::getStopMask`, examining the ori-

gins behind a debugger caused pause, managing breakpoints and other debugger variables, and similar.

4. Inspecting machine state: the `Machine` class exports methods for reading and writing memory, and retrieving processor and device model instances.

4.3 The User Interface Implementation

In the following sections, the implementation of µMPS2 user interface (UI) is described. Our discussion will consists of three parts that focus on rather different aspects. First, the basic architectural style—in part dictated by the Qt framework—is described. Next, we look at the manner in which virtual machine execution is integrated with other tasks (e.g., the application's main event loop). Finally, a dissection of the UI components into classes and modules is given.

4.3.1 The Qt Framework

The µMPS2 user interface is built using Qt, a cross-platform graphical user interface toolkit and, more generally, a fully-fledged C++ application framework. Along with GTK+, it is currently one of the few inherently cross-platform widget toolkits for modern desktop systems. In addition to the GUI library, it contains modules in a wide variety of categories—from simple abstractions of common platform facilities (e.g. thread support) to database access and network programming.

A feature of Qt 4 that is leveraged to a great extent within µMPS2 is its model-view-controller architecture [24, Chapter 5] (referred to as *Interview* in Qt jargon) for item views—i.e., lists, tables, and (multi-column) trees. More details on this are given in the following sections.

Needless to say, an introduction to Qt would not be within the scope of this work; the interested reader is advised to consult one of the many introductory books and references available. In particular, [25] covers in great detail the more advanced aspects of Qt 4.x, such as its model-view architecture.

4.3.2 Models and Views

Much of the user interface code in µMPS2 is structured around the *model-view-controller* (or more simply, *model-view*) paradigm [24, 26]. This is in part because this architectural pattern has proven to be a sound choice (especially for applications featuring a GUI), and in part because it is a style that is encouraged by (and to a degree dictated by) the Qt framework. The essence of this architectural style is characterized by a form of separation of concerns: *models* serve as content providers and encapsulate the relative data-processing rules; *views* are visual representations of the content; *controller* code serves as a mediator between user input, the model, and the view.

There are many ways in which the model-view architecture fits the interaction between user interface and emulator back-end components, as shown schematically in figure 4.2:

- The machine-level abstractions provided as part of the core emulator API (`Machine`, `Processor`, and similar) would themselves be considered models, of course. Moreover, virtually all other models are built on top of these.

- Qt defines numerous standard abstract model interfaces in its Interview framework. Each of these interfaces defines a set of APIs a particular data source must provide for interoperability with one or more types of views. For instance, the `QAbstractTableModel` class defines an abstract model interface implemented by classes that (conceptually) represent a rectangular array of data elements of a particular kind (numbers, strings, icons, etc.).

  There are several such implementations of the Qt Interview model interfaces in μMPS2. For example, processor registers in the dockable pane of the processor window are rendered by a `QTreeView` object, one of the framework's many *view* classes; the model is, of course, not a stock Qt model class but a custom implementation of the generic *tree model* interface, its elements being register mnemonics and register values (see `RegisterSetSnapshot`). The values are in turn obtained from the `Processor` instance itself. We may call such classes *model adapters*.

  Another notable example of a model adapter class is `StoppointListModel`, a Qt *table model* adapter used to represent breakpoint, suspect, and tracepoint collections.

- View classes include both standard Qt classes and custom ones. Stock Qt view classes are typically used in conjunction with the model adapter classes discussed above, whereas custom view classes interact with core emulator models and with various auxiliary models.

4.3.3 Machine Execution

Modern graphical user interface toolkits impose on the program an event-driven programming model. In most cases, this model is supported by providing a form of *event loop* abstraction[7], which manages all low-level aspects associated with event sources of interest to the application. In a typical program flow, suitable *event handlers* are registered for particular events, and control is handed over to the event loop. The event loop blocks until one or more of the specified events are detected—for example, those originating from the underlying window system—and then dispatches

[7]Slight exceptions are the Swing and AWT toolkits; rather than defining an explicit event loop construct, these toolkits dedicate a special thread to event polling and dispatching. The "event loop" is still present, of course, but it is completely opaque to the programmer.

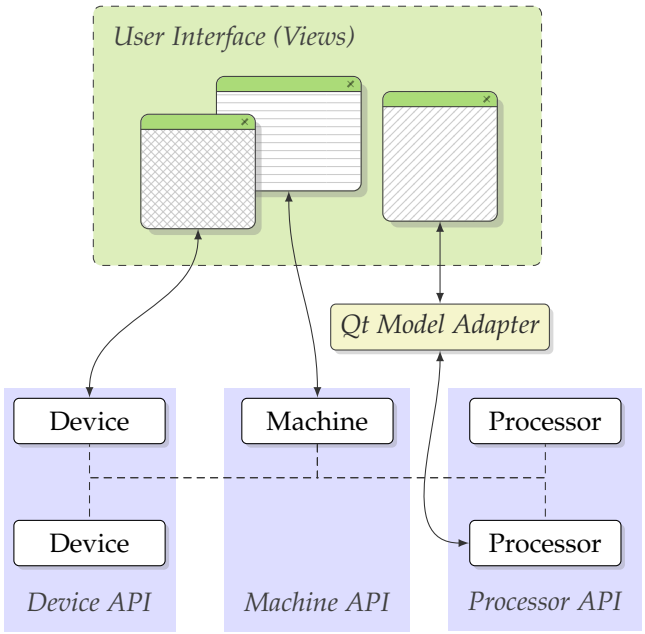


Figure 4.2: *The model-view architecture of the µMPS2 emulator.*

these to applicable handlers. After all handlers have been invoked, control is returned to the event loop and another iteration of the cycle is started.

Due to difficulties in ensuring thread safety of their respective APIs, GUI toolkits have traditionally been inherently single-threaded, with Qt not being an exception to this.[8] More precisely, all user interface related processing is confined to a single thread of execution; the thread, that is, in which the UI event loop is run. All of this places an important policy constraint on event handlers: to ensure proper responsiveness of the UI, a handler is not expected to perform any long-running task (e.g., long-running computations or certain kind of I/O) directly. It is immediate to see that virtual machine execution in µMPS2 is, in fact, an example of such a compute-bound task. An often-used solution for this problem consists of offloading the computation to a separate thread or process. For the reasons stated above, this technique is most-appropriate for cases in which the task is not required to perform a significant amount of manipulations on the UI itself. Another popular approach consists of splitting the computation into smaller parts, and schedule them for execution via the event loop as low priority tasks; the event loop will process the tasks, one by one, whenever there are no higher priority events pending (notably, window system events). This latter approach is currently used in µMPS2, as the risk of concurrency-related issues was considered to significantly outweigh the slight increase of control flow complexity.

All aspects of machine execution are handled by the `DebugSession` class. In particular, it provides emulation speed control. The effective emulation speed (that

[8] The difficulties, in short, stem from the inherently asynchronous nature of GUI applications. The interested reader is referred to [27] for a discussion on this topic.

is, the rate at which machine cycles are executed) is determined by several variables we describe below, each of which is derived from the user-controlled speed parameter.

In continuous execution mode (as opposed to single-stepping mode), emulation is performed in the fashion described above: the task is performed incrementally, and in each *iteration* a number of machine cycles are executed; the exact number of cycles per iteration is determined by the emulation speed setting. Idle machine cycles are properly skipped, as allowed by the underlying emulation engine. For perceived realism, the number of skipped cycles is accounted for by a suitable pause in machine execution. As a result, an idle virtual machine is reflected in an idle emulator process.

The *maximum* effective rate at which machine cycles can be run is further controlled by the *minimum* delay between two successive machine cycle iterations. Again, this value is derived from the user-controlled speed setting. Incidentally, the degree to which the emulator will be a CPU-bound process is directly related to this variable.

Finally, the `DebugSession` provides suitable notification—most importantly, to UI components—of a number of related events:

- *The machine is powered on, powered off, or reset*. Some UI elements are either created or initialized at machine power-on completion. Examples of such elements are the processor status list and device status view in the main window. Likewise, after the machine has been powered down, these elements are dissociated from the machine state they are representing, and thus need to be destroyed or otherwise invalidated. Reset notification is provided since virtually all visible UI elements pertaining to machine state need to be updated on that event.

- *Machine execution is suspended or resumed*. Certain UI elements, such as the code view, are only sensitive to user input and exhibit meaningful semantics while emulation is paused. (Indeed, in the case of the code view, a "live update" of the widget while the machine is run would not only be distracting to the user, but it would also incur a significant deterioration of performance.)

  Similarly, actions such as breakpoint insertion and removal can only be performed while emulation is paused. Thus, their associated user interface elements (menu bar entries, context menu entries, toolbar buttons, etc.) are rendered sensitive (i.e., the elements are "enabled") if and only if the machine is stopped.

- *An iteration of machine cycles is completed*. Some machine state views update their contents even while emulation is in progress. Since updates on each cycle would be unfeasible, the refresh is instead triggered by this event. The number of cycles between each update is thus dependent on the user's emulation speed setting.

Elements that support this form of live updating include processor status entries, processor register views, traced memory regions, and the device status view. Of course, displayed machine state is *guaranteed* to be "consistent" (that is, reflect actual state values) only while the machine is stopped.

Notification is provided using Qt's signaling mechanism; for instance, the *machine reset* event is represented by the `MachineReset` signal (see `debug_session.h`) and is intercepted by any UI component that need to act on this event.

4.3.4 Class and Module Overview

After the above overview of architectural aspects of the µMPS2 front-end, what follows is a fairly complete outline of the UI code base. Rather than listing classes or modules in some arbitrary order, we systematically group classes by their user interface affinity, starting from top level windows:

- *Main window*. The main window is represented by the `MonitorWindow` class.[9] While most sub-panes in this window are implemented by separate classes, the class alone manages a fair number of different aspects: its tab based layout, the `QAction` based menu and toolbar interface, auxiliary window visibility, and many other ancillary tasks.

  The greater part of the main window is occupied by its tabbed sections:

  - *Machine overview tab*. This tab is implemented by `MachineConfigView`, a simple composite widget that displays machine parameters in a table-like layout.
  - *Processor tab*. This tab is split into an upper half, which contains the processor status list, and a lower one, which contains the breakpoint list. The effective tab area allocated to the two child widgets is user controllable and is managed by the `QSplitter` widget.

    The underlying model class for the processor status list (or, more precisely, *table*) is `ProcessorListModel`; the latter is a direct subclass of Qt's `QAbstractTableModel` abstract class, which defines an interface expected to be provided by all table model classes within the Interview framework. The view itself is rendered by the stock `QTableView` class.

    An entirely analogous separation of concerns—those of a model from those of a view—applies for the breakpoint, suspect, and tracepoint lists. In this case, the model is provided by the `StoppointListModel` class.
  - *Memory tab*. The memory tab contains the suspect list and a *traced region browser*. The former is completely analogous to the breakpoint list discussed above.

[9] As a general guideline, there is a direct correspondence between a class and a (single) translation unit, with a slight difference in naming convention between the two. An example will suffice: the class `TerminalView` is associated with the class implementation file `terminal_view.cc` and the corresponding header file `terminal_view.h`.

As discussed in Section 3.4.5, multiple display types are supported for traced memory regions. The associations between traced region and the user's preferred display type is managed by the `TraceBrowser` class. Display-wise, `TraceBrowser` is a composite widget comprising the tracepoint list, the display type drop-down list (frequently called a "combo box"), and the memory view itself.

The ASCII rendition of the memory is rather simple (see the `AsciiView` class in `trace_brower.h`) and should not require explanation. As expected, considerably more complex is the editable hex-dump view implementation, given by `HexView`. By subclassing `QPlainTextEdit`, Qt's plain text visualization and editing widget, it leverages the widget's extensive text display and manipulation facilities (e.g., text highlighting and scrolling support). At the same time, the hex view requires customized key and mouse-based navigation within the grid-like display. Similarly, since the memory itself acts as a "backing store" for its contents, editing requires special handling. Support for these is implemented by overriding certain `QPlainTextEdit` methods (see the `HexView::keyPressEvent` method, for an example).

– *Device status*. Again, the same split between the view and underlying model applies (as in the case of the processor list, for instance). In this case, the implementation (`DeviceTreeModel`) is considerably more complex, since the model's data retrieval mechanisms must support a hierarchical structure, as specified by the `QAbstractItemModel` interface.

- *Processor window*. Top-level processor windows (`ProcessorWindow`) feature a central content, consisting of the code view and processor status information, and certain secondary panes. At the user's preference, these secondary windows can either be positioned inside the processor window around its central content or made into top-level windows. Secondary windows are implemented using Qt's *dock widgets* (see `QMainWindow` and `QDockWidget`). Details about the main elements follow:

  – *Code view*. The code view (`CodeView`) is another subclass of Qt's plain text display and editing widget, `QPlainTextEdit`. The implementation provides a custom margin widget (see `CodeViewMargin`) onto which memory addresses, breakpoint markers, and the program counter marker are rendered (see `CodeView::paintMargin`). Breakpoint insertion and removal is supported by the margin by overriding mouse event handlers.

  – *Register view*. The register view is one of the two "dockable" widgets belonging to the processor window. The top-level widget itself, namely `RegisterSetWidget`, manages the selection of register value representation format (e.g., binary, decimal, etc.).

  Register items (that is, mnemonic and corresponding value pairs) themselves are rendered by the `TreeView` widget. The underlying data model

is provided by `RegisterSetSnapshot`—yet another example of a Qt Interview abstract item model realization.

Recall that the register view is one of the elements periodically updated even while the machine is running. In order to support this reasonably efficiently, `RegisterSetSnapshot` implements a form of "lazy" update notifications. Register values fetched from the CPU are cached on each refresh cycle, so that on the following cycle new values can be compared with cached ones, and the view can as a result be only notified about modified values.

- *TLB view*. This is essentially similar to the register view described above (see the `TLBModel` class).

- *Terminal window*. Terminal windows are instances of `TerminalWindow`; the actual widget implementing a text terminal is `TerminalView`—another subclass of `QPlainTextEdit`. Where needed, the `TerminalView` class overrides its superclass method implementations to simulate the behavior of a real text terminal (see `TerminalView::keyPressEvent`, for example).

4.4 μMPS2 Object File Support Tools

For reasons that have already been elaborated (see Section 3.5.3), μMPS2 includes support for a greatly streamlined executable file format. Two object file-related tools are currently available: `umps2-elf2umps` converts ELF relocatable objects and executables to μMPS BIOS images and .aout/.core executables, respectively; `umps2-objdump` is an analogue of the well known objdump utility, albeit less sophisticated. These tools are provided as stand-alone executables—rather than, say, being integrated into the GUI environment—the reason being that they are typically expected to be used as part of an automated build workflow.

Among executable file formats, .aout is exceptionally simple. For reasons of space, its specification is not replicated here; the reader is referred to [11]. In the rest of this section, however, the most important aspects of the ELF to .aout conversion are addressed, including some rather subtle points.

4.4.1 ELF to .aout Conversion

The ELF to .aout conversion code (see `elf2umps.cc`) is fundamentally not very complex. (Needless to say, a working understanding of the conversion process demands at least basic familiarity with the ELF format, for which we refer the reader to [28] or [29]). Essentially, ELF input *sections*[10] are scanned, one by one, and data from

[10]There is some terminological inconsistency across object file formats regarding the interpretation of "section" and "segment". While these two words are often used interchangeably, in the context of the ELF format they denote different concepts. For reasons mostly based on efficiency and flexibility, ELF supports a dual view of an object file. First, a file may be treated as a collection of *sections*—that is, units containing program data or metadata of a particular kind. Thus, code, data, read-only data,

"loadable" sections is copied to the corresponding *segment* of the .aout executable (see the `elf2aout` function). The simplicity stems from an important assumption about the input ELF executable; namely, that its memory layout (the *segment layout*) matches the desired memory layout of the corresponding .aout executable. This requirement is satisfied by providing suitable directives to the linker (`ld`), in the form of a *linker script*. In short, a linker script defines the manner in which sections from one or more input object files are combined into sections of the output file. In doing so, the script defines the memory layout of the output file.

We see that in effect, an .aout executable is obtained via a cooperation of the linker—driven by a linker script—and the `umps2-elf2umps` utility itself. We wish to elaborate somewhat on the former, since the linker script command language may admittedly seem arcane to the uninitiated. (For a detailed description of the linker command language accepted by the GNU linker, the reader is referred to its official documentation. Barely enough syntax is introduced here to render the discussion comprehensible.)

Listing 4.2 shows a linker script suitable for .aout executables. Any linker scripts consists of a series of commands, each of which controls a particular aspect of the linking process. An example is the `ENTRY` command on line 2, which defines the program's entry point.

The `PHDRS` command (lines 4-8) lists the *program headers* (segments) the output executable should contain. Those will correspond to the *.text* and *.data* segments in the final .aout executable.

The rest of the script is taken by the compound `SECTIONS` command. Of all commands allowed inside `SECTIONS`, two types are encountered in the script below:

- *Symbol assignments*: Symbols may be defined and assigned arbitrary values using linker scripts. These symbols will appear in the resulting object's symbol table, just like those generated by the assembler. For example, on line 34 the symbol `_gp` is defined.

  As a special case, the symbol name '.', called the *output location counter*, denotes the memory address the next portion of the output is normally associated with.

- *Output section descriptions*: These commands describe how input sections should be merged to form output sections. The full syntax of an output section description command is rather complicated, and since most occurrences do not need nor use most of the features, it suffices for our needs to examine the first such command appearing in the script below (lines 14-18). The `.text` identifier outside of the curly braces is the *output section name*. Inside the curly braces, the most important command that may appear is an *input section description* (line 17), which causes selected input sections from one or more input files to be copied to the output section in question (in this case, `.text`).

relocation information, debugging information, etc. all belong to different sections of the file. The file is endowed with this structure to support linking, in addition to tools such as debuggers. On the other hand, an ELF executable file may be viewed as a set of *segments* that define its memory layout. This second view exists to support the run-time system (loaders) efficiently.

Each input section description includes an input file name followed by a list of section names in parenthesis, both of which can use Bourne shell-like wildcard patterns. The command from our example, namely `*(.text .text.*)`, instructs the linker to merge into the current output section all sections matching the names `.text` or `.text.*` from any input file.

After this brief demystification, it is now hopefully easy to see how the mentioned linker script results in a memory layout that complies with the µMPS .aout format requirements. The base address of the .aout .text area is `0x8000.0000`; to accommodate the 90-byte .aout header, the sections comprising the *text* segment of the ELF executable start at `0x8000.00B0` (line 13). The text segment contains all code and read-only data sections. The data segment starts on the first page-aligned address after the text (line 27); it contains the .data sections, the global offset table (GOT) [28], the "small data sections", and similar.

Listing 4.2: A linker script for .aout executables.

```
1  OUTPUT_ARCH(mips)
2  ENTRY(__start)
3
4  PHDRS
5  {
6      text PT_LOAD FILEHDR PHDRS;
7      data PT_LOAD;
8  }
9
10 SECTIONS
11 {
12      /* Text segment */
13     . = 0x800000B0;
14     .text :
15     {
16         _ftext = . ;
17         *(.text .text.*)
18     } :text =0
19     PROVIDE (_etext = .);
20
21     .rodata : { *(.rodata .rodata.*) } :text
22     .rodata1 : { *(.rodata1) } :text
23     .sdata2 : { *(.sdata2 .sdata2.*) } :text
24     .sbss2 : { *(.sbss2 .sbss2.*) } :text
25
26     /* Data segment */
27     . = ALIGN(0x1000);
28     .data :
29     {
30         _fdata = . ;
31         *(.data .data.*)
32     } :data
```

```
33      .data1 : { *(.data1) } :data
34      _gp = ALIGN(16) + 0x7ff0;
35      .got : { *(.got.plt) *(.got) } :data
36      .sdata : { *(.sdata .sdata.*) } :data
37      .lit8 : { *(.lit8) } :data
38      .lit4 : { *(.lit4) } :data
39      _edata = .;
40
41      __bss_start = .;
42      _fbss = .;
43      .sbss :
44      {
45          PROVIDE (__sbss_start = .);
46          *(.sbss .sbss.*)
47          *(.scommon)
48          PROVIDE (__sbss_end = .);
49      } :data
50      .bss :
51      {
52          *(.bss .bss.*)
53          *(COMMON)
54         . = ALIGN(32 / 8);
55      } :data
56
57      . = ALIGN(32 / 8);
58      _end = .;
59      PROVIDE (end = .);
60
61      /DISCARD/ : { *(.reginfo) }
62  }
```

Chapter 5

Conclusions

The μMPS system, which formed the basis for the work that has been described in this thesis, was designed to fulfill the need for an approachable and at the same time reasonably realistic computer architecture and accompanying emulator, tailored for use in computer science education.

The most important way μMPS2 improves upon its predecessor is through the architecture update. By including multiprocessor support, μMPS2 renders the architecture considerably more relevant in an era in which multi-core designs are becoming ever more prevalent. It is hoped that, by virtue of being exceptionally simple implementation-wise, the system will be able to adapt equally promptly to future hardware trends.

The other major overhaul in μMPS2 is in the emulator's front-end, which (it is hoped) has resulted in notable usability improvements. The μMPS2 user interface environment is currently unique in its category in providing intrinsic multiprocessor monitoring and debugging support.

5.1 Suggestions for Further Experiments

Some suggestions for possible extensions to μMPS are given below. Needless to say, the list is not in any case exhaustive, nor are those necessarily the most viable directions for the project. As many users are bound to discover, the current system is surely open to many smaller, incremental, improvements.

5.1.1 Detailed Simulation

A direction arguably worth pursuing with μMPS concerns more detailed and realistic simulation models, particularly with regard to the memory system. Indeed, that (centralized or distributed) shared memory architectures will have an ever more profound impact on software design in the near future—both for operating systems and end-user applications—is suggested by current trends, and is likely not a reckless prediction.

The simple CPU model in μMPS2 could be combined with realistic memory system models, including MESI-based coherent caches, cache coherent NUMA, and similar. Of course, it would be most desirable to continue to support the currently used simpler emulation model, for cases where quick execution is a priority over realism.[1]

5.1.2 Emulator Scalability

The emulator implementation described in Chapter 4 has multiple virtues, including simplicity of implementation and a fully deterministic execution model. Unfortunately, the implementation in its current form is inherently non scalable beyond a small number of processors, as it uses a single thread of execution to emulate multiple virtual processors in lockstep fashion. A currently investigated alternative design consists of exploiting thread-level parallelism at the *host level*, the key idea behind this design being that various components comprising a computer system are to a great extent autonomous—that is, largely independent of each other.

In the proposed experimental implementation, a dedicated thread is assigned to each emulated processor; a separate thread executes the emulator event loop, which handles event scheduling (e.g., expiration of various hardware timers and I/O completion) and dispatching to event handlers. Long-running blocking operations may be delegated to special worker threads, allocated from a thread pool.

One consequence of introducing this form of thread-based concurrency into the emulator is that the architecture's intuitive, strictly ordered memory consistency model (see [31] for a discussion of memory consistency models in general, and ordering semantics in particular) must in all likelihood be sacrificed. Indeed, to have reasonable chance of gaining an advantage from the introduced parallelism, the emulator cannot guarantee ordering semantics stricter than those of the underlying host's memory system. A consistency model must then be defined for the emulated architecture whose semantics correspond to the weakest model among the supported host architectures. Next, a set of memory barrier instructions must be provided, whose implementation will naturally depend on the particular host architecture.

Finally, changes are required in virtual time management. In the single threaded implementation virtual processors run in lockstep manner and single processors cycles correspond to ticks of the centrally maintained "time-of-day" clock. This clock is maintained by the emulator and has no bearing on host time. In a multithreaded implementation where, at least to some extent, each virtual CPU runs independently of the rest of the system this is not feasible anymore. Instead, it is desirable to base virtual time bookkeeping on one of the high resolution clock sources provided by the host system.

[1] For reference, a system that supports varying degrees of simulation speed and detail, from fast execution modes based on dynamic translation techniques to detailed microarchitecture models, is SimOS [30].

5.1.3 An Operating System for μMPS2

All testing has so far relied on simple, mostly ad hoc "operating systems". A reasonably featured, SMP-capable operating systems would undoubtedly prove very useful during emulator development (e.g. for tasks such as performance evaluations or regression tests). One could either develop such an OS from scratch, or choose to port one of the lightweight open source operating systems to μMPS2. Both options promise an educationally rewarding experience.

Appendix A

μMPS2 Architecture Revision

This appendix presents the new hardware and firmware features of μMPS2. The prominent new features center around multiprocessor support: initialization and control of multiple processors, interrupt management in a multiprocessor environment, new processor instructions for concurrency support and ROM services. Wherever possible, architectural additions were designed to be unobtrusive with regard to legacy software and programmers not striving to support software features (such as SMP support) that require them.

A.1 Machine Control Registers

μMPS2 provides the programmer with explicit control over the power states of CPUs and the machine itself. This is accomplished through the *Machine Control* register set, shown in Table A.1.

The **NCPUs** register stores the number of processors in the system; **ResetCPU**, **BootPC**, **BootSP**, **HaltCPU** control the processors power state and start-up (state on reset); finally, the **Power** register is used to power off the machine.

Table A.1: *Machine control registers address map.*

Address	*Register*	*Type*
0x1000.0500	**NCPUs**	Read Only
0x1000.0504	**ResetCPU**	Write Only
0x1000.0508	**BootPC**	Read/Write
0x1000.050c	**BootSP**	Read/Write
0x1000.0510	**HaltCPU**	Write Only
0x1000.0514	**Power**	Write Only

A.1.1 Processor Power States

At each point in time a processor in µMPS2 can be in one of several *power states*, which define whether it is currently executing instructions and its responsiveness to external events (interrupt, reset and halt signals).

µMPS2 defines three power states:

- *Halted*: This state represents the lowest power state. A CPU in this state will only respond to a *reset* signal, which transitions the CPU into the *Running* state, causing it to start executing instructions.

  A CPU transitions into this state when its *halt* signal is asserted, which is triggered by writing the CPU ID into the **`HaltCPU`** register. The CPU does not maintain any architecturally visible state (e.g. CPU registers) in this power state.

- *Running*: This state represents the normal operating state of the CPU. A CPU in this state responds to both interrupts and halt/reset signals. A CPU transitions into this state as a result of external events.

- *Idle*: A CPU in this state operates in reduced-power mode. The CPU stops executing instructions when it transitions into this state, but it stays responsive to all external events. A CPU transitions into this state by executing the `wait` instruction. The CPU maintains all architecturally visible state in this power state. This state is also often referred to as *standby*.

Figure A.1 recaps the possible transitions between power states.

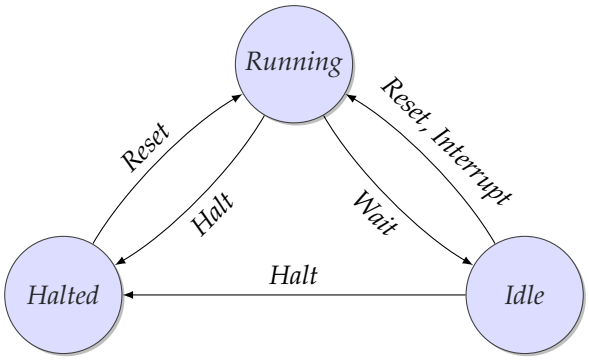


Figure A.1: *Processor power states.*

A.1.2 Processor Initialization

After a machine reset, only processor 0 is automatically started (i.e. in the *Running* power state). Explicit start-up (*reset*) commands must be issued to start the other processors. A secondary processor starts executing when it receives a *reset* signal. This is accomplished by writing the processor ID into the **`Reset`** register. The processor

starts executing at the location specified by the **BootPC** register, with the processor's **sp** register set to the value provided by the **BootSP** register. All other aspects of the processor state at reset time are as described in [11].

As described in Section A.5, the BIOS and `libumps` library provide a more convenient way to initialize a processor.

A.1.3 Powering Off the Machine

Machine power off is initiated by writing the magic value `0x0FF` into the write-only **Power** register. The power down completes after a non-negligible delay.

A.2 New and Revised CP0 Registers

Among the software-visible changes introduced with μMPS2 some are within the system control coprocessor's (CP0) register space. Table A.2 lists CP0 registers along with compatibility notes. The following sections describe each change in detail and motivate their introduction.

Table A.2: *CP0 registers.*

Mnemonic	*Number*	*Compatibility Notes*
Index	0	-
Random	1	-
EntryLo	2	-
BadVAddr	8	-
Timer	9	New in μMPS2
EntryHi	10	-
Status	12	A new read/writable **TE** field has been added (see Figure A.2).
Cause	13	**IP[0]** and **IP[1]** are no longer writable.
EPC	14	-
PRID	15	This read-only register now stores the processor ID instead of CPU model information.

A.2.1 PRID Register

In μMPS2 the **PRID** register contains the *processor identifier* (CPUID), which is a unique integer assigned to a processor at power up that does not change afterwards.

The read-only **PRID** register has been used in MIPS CPUs as a *CPU model identifier*. In contemporary MIPS processors, it contains a field that identifies the instruc-

Table A.3: *Interrupt line assignment in µMPS2.*

Interrupt Line	*Assignment*
0	IPI
1	Local Interval Timer
2	Bus (Interval Timer)
3	Disk Devices
4	Tape Devices
5	Network (Ethernet) Devices
6	Printer Devices
7	Terminal Devices

tion set architecture, along with manufacturer dependent fields. Since such information is only plausibly useful to legacy MIPS software which interpret it (typically as part of the hardware probing process) and µMPS implements a fixed variant of the MIPS I ISA, the above redefinition of the **PRID** register is very unlikely to cause compatibility problems.

A.2.2 The On-CPU Interval Timer

µMPS2 provides a per-CPU interval timer (referred to as the *local timer* in this section) that runs at the processor's clock rate and is in all respects similar to the external interval timer device. Unlike the external interval timer, the local timer can be disabled (default setting). Whether the local timer is enabled or not is determined by the **Status.TE** (*Timer Enable*) bit. When **Status.TE** $= 0$, the local timer will not generate interrupts. Note, however, that it is implementation dependent whether the timer will continue to run when **Status.TE** $= 0$. If the local timer is enabled (**Status.TE** $= 1$), it will continue to run and generate interrupts even when the processor enters standby mode (see Section A.4.2).

Interrupts from he local timer are assigned to interrupt line 1 (see Table A.3). The CP0 **Timer** register represents the programmer's interface to the local interval timer. Pending interrupts from the local timer are acknowledged by writing a new value into the **Timer** register.

The local interval timer will most likely only be of use to operating systems with multiprocessor support, where the local timer can be used for scheduling purposes. Since interrupts from the external timer source can only be delivered to a single CPU at a time, this can potentially avoid the overhead of interprocessor interrupts.

A.2.3 Status Register

The **Status** register contains a single new field, **TE**, that enables/disables the on-CPU timer.

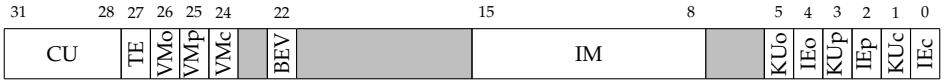


Figure A.2: *The* **Status** *CP0 register.*

A.2.4 Backward Compatibility Notes

Two of the described modifications of the CP0 register space potentially compromise backward compatibility with existing software. As already noted above, the redefinition of the **PRID** register is unlikely to cause compatibility problems.

Interrupt lines 0 and 1, previously reserved for software interrupts, are reassigned to inter-processor interrupts and the on-CPU timer, respectively. Programs that relied on this older form of software generated interrupts will find *inter-processor interrupts* (described in a separate section) to be a relatively straightforward replacement.

A.3 Multiprocessor Interrupt Control

The new interrupt delivery subsystem in μMPS2 is designed to support SMP-capable operating systems. It allows the creation of elaborate interrupt affinity and/or balancing schemes and provides a simple *inter-processor interrupt* (IPI) mechanism. Note that the default settings of the registers described in this section provide for full backward compatibility with uniprocessor systems. Thus, the register-level interface described here will only be of concern to programmers willing to support multiprocessor systems.

Conceptually, the multiprocessor interrupt controller is comprised of the following units:

- A centralized programmable unit called the *Interrupt Router* that distributes interrupts from peripheral interrupt sources to selected processors.
- One or more *CPU Interface* units that receive interrupts from the Interrupt Router and control the transmission and reception of IPI messages.

A.3.1 Interrupt Distribution

For systems under heavy I/O load, it is often desirable to distribute interrupts across multiple processors. μMPS2 allows the programmer to specify interrupt routing

parameters per interrupt source. Routing parameters are stored in a set of programmable registers, called the *Interrupt Routing Table* (**IRT**). Each IRT entry controls interrupt delivery for a single interrupt source.

Two distribution policies are supported:

- *Static:* The interrupt is delivered to a preselected processor.
- *Dynamic:* The interrupt is delivered to the processor executing the lowest priority task.

Figure A.3: ***IRT*** *entry format.*

Each IRT entry register (see Figure A.3) consists of the following fields:

- **RP** (bit 28): Specifies the routing policy. The field is interpreted as follows:

  0 (Static) The interrupt is delivered to the processor specified in the **Destination** field. The **Destination** field is interpreted as a CPU ID.

  1 (Dynamic) The **Destination** field is interpreted as a CPU mask, where bit *i* of **Destination**`[15:0]` corresponds to CPU ID *i*. The interrupt is delivered to the processor executing the lowest priority task among all contestants selected by the mask. In case of a tie, an implementation-defined arbitration mechanism is used to resolve it.
  The above mechanism relies on the operating system to update at appropriate times the execution priority of all selected processors. This is accomplished by programming the *Task Priority* (**TPR**) register, located in the interrupt controller CPU interface register bank (see section A.3.2).

- **Destination** (bits 0-15): Used to specify the interrupt target processor(s). See the description of the **RP** field for details on how this field is interpreted.

As shown in figure A.4, the IRT has 48 entries. Interrupt routing information for device device j, attached to interrupt line i, is recorded in entry $(i-2) \times 8 + j$.

A.3.2 CPU Interface Registers

The CPU Interface registers, shown in Table A.4, represent the per-processor component of the multiprocessor interrupt controller register-level interface. Although multiple banks (one per CPU) of these registers are provided, they all share the same address map.

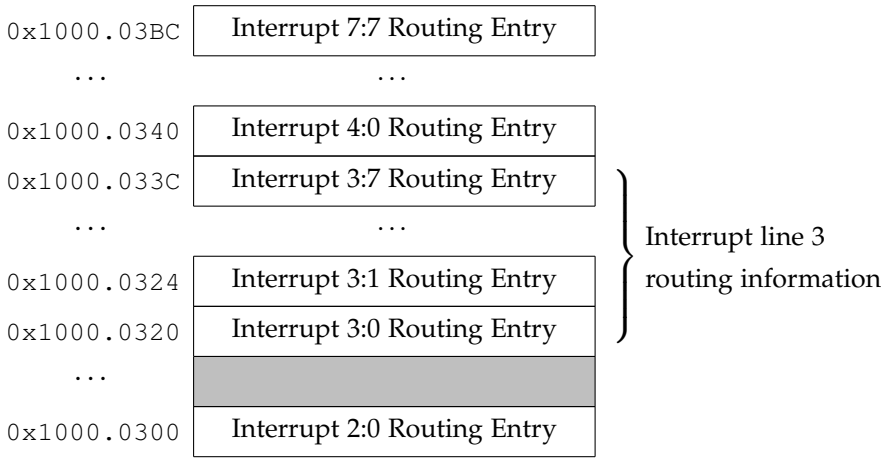


Figure A.4: ***IRT*** *register address map.*

Table A.4: *Interrupt controller processor interface register map.*

Address	*Register*	*Type*
0x1000.0400	**Inbox**	Read/Write
0x1000.0404	**Outbox**	Write Only
0x1000.0408	**TPR**	Read/Write
0x1000.040c	**BIOSReserved1**	Read/Write
0x1000.0410	**BIOSReserved2**	Read/Write

The **Inbox** and **Outbox** registers are used for inter-processor interrupts and their use is detailed in the next section.

The *Taks Priority* (**TPR**) register, shown in Figure A.5, is used by the Interrupt Router in the priority based arbitration scheme. The **TPR.Priority** field allows for 16 priority levels, with 0 and 15 representing the highest and lowest priorities respectively.

Figure A.5: *The* ***TPR*** *register.*

The two registers labelled as *BIOS Reserved* are provided for the convenience of the ROM based exception handling code.

A.3.3 Inter-processor Interrupts (IPIs)

An inter-processor interrupt (IPI) represents a form of inter-processor signaling mechanism used by a processor to request the attention of another processor. IPIs are

commonly used by operating systems for issuing rescheduling requests, maintaining TLB consistency, and any other task which requires one processor to request the attention of another.

In µMPS2, multiple IPIs may be pending at the same time for a given processor; there is, however, a limit of *one pending IPI per originating processor*. Only one pending IPI may be acknowledged at a time. There is no built-in delivery status notification mechanism.

An arbitrary 8-bit data field (*message*) can be carried with each IPI. This feature is provided solely for software convenience and has no side effects on the IPI delivery subsystem.

The system maintains IPI delivery order. IPI messages are always retrieved in the order they were received by the CPU interface.

Issuing IPIs

An IPI is issued by writing a correctly formatted IPI command to the **`Outbox`** register, shown in Figure A.6.

Figure A.6: *The* ***`Outbox`*** *register.*

The fields in the **`Outbox`** register are defined as follows:

- **`Message`** (bits 0-7): The message to be delivered.
- **`Recipients`** (bits 8-23): Selects the recipients of the IPI. The field is interpreted as a CPU mask: the IPI is delivered to processor *i* if **`Recipients[`***i***`]`** is on.

IPI Receipt and Acknowledgement

Information on the currently pending IPI is stored in the **`Inbox`** register (shown in Figure A.7) which contains the following fields:

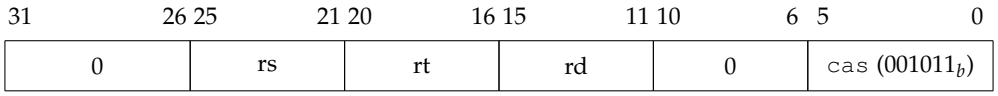
- **`Message`** (bits 0-7): The message to be delivered.
- **`Origin`** (bits 8-11): The processor ID of the originating processor.

Figure A.7: *The* ***`Inbox`*** *register.*

An IPI is acknowledged by writing to the **`Inbox`** register. The written value is ignored.

A.4 New Instructions

A.4.1 Compare and Swap (CAS)



Format:
`cas rd, rs, rt`

Description:
The `cas` instruction performs an atomic read-modify-write operation on synchronizable memory locations. The contents of the word at the memory location specified by the GPR **`rs`** is compared with GPR **`rt`**. If the values are equal, the content of GPR **`rd`** is stored at the memory location specified by **`rs`** and 1 is written into **`rd`**. Otherwise, 0 is written into **`rd`** and no store occurs.

The above read-modify-write sequence is guaranteed to be *atomic* by ensuring that no intervening operation on a conflicting memory location is performed by the memory system. The following pseudocode illustrates the operation of the `cas` instruction:

```
atomic {
    if (MEM[rs] == GPR[rt]) {
        MEM[rs] = GPR[rd];
        GPR[rd] = 1;
    } else {
        GPR[rd] = 0;
    }
}
```

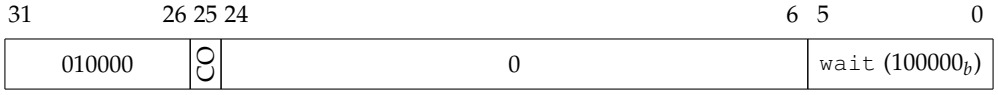
The set of *synchronizable memory locations* in μMPS2 coincides with physical RAM locations. For all other locations (e.g. the I/O address space) `cas` will unconditionally fail.

Exceptions:
TLBS, Mod, DBE, AdES

libumps interface:
`int CAS(uint32_t *atomic, uint32_t ov, uint32_t nv)`

This function atomically sets the word pointed to by `atomic` to `nv` if the current value of the word is `ov`. It returns 1 to indicate a successful update and 0 otherwise.

A.4.2 Wait for Event (WAIT)



Format:
`wait`

Description:
The `wait` instruction causes the processor to enter standby mode. The processor resumes execution when an external event (reset or interrupt) is signaled to the processor. Note that it is irrelevant whether the signaled interrupt is enabled or not. If the processor resumes execution as a result of an *enabled* interrupt and **`Status.IEc`** is on, the (interrupt) exception is considered to have occurred at the instruction following the `wait` instruction.

Exceptions:
CpU

libumps interface:
`void WAIT(void)`

A.5 BIOS Services

The BIOS ROM code supplied with µMPS provides convenient low-level services an OS can rely on or build upon. Most importantly, the general and TLB-refill exception entry points lie in the ROM space. In µMPS2, the BIOS exception handling code has been made reentrant with regard to multiple processors. To ensure reentrancy, separate *Old* and *New Processor State Areas* must be provided for each processor.

A new BIOS service has been added to hide most of the complexities of processor startup and initialization of BIOS-related processor data structures. It is exposed by the libumps library function `INITCPU`:

```
void INITCPU(uint32_t cpuid, state_t *start_state, state_t *state_areas);
```

This function initiates a reset of the processor specified by `cpuid`, causing it to start execution at a preselected startup entry point in the ROM. The startup routine initializes the BIOS data structures related to the processor (most importantly, it records the address of the New/Old State areas, given in the `state_areas` parameter). Finally, it loads the processor state from the supplied `start_state` parameter.

A.6 Device Register Memory Map

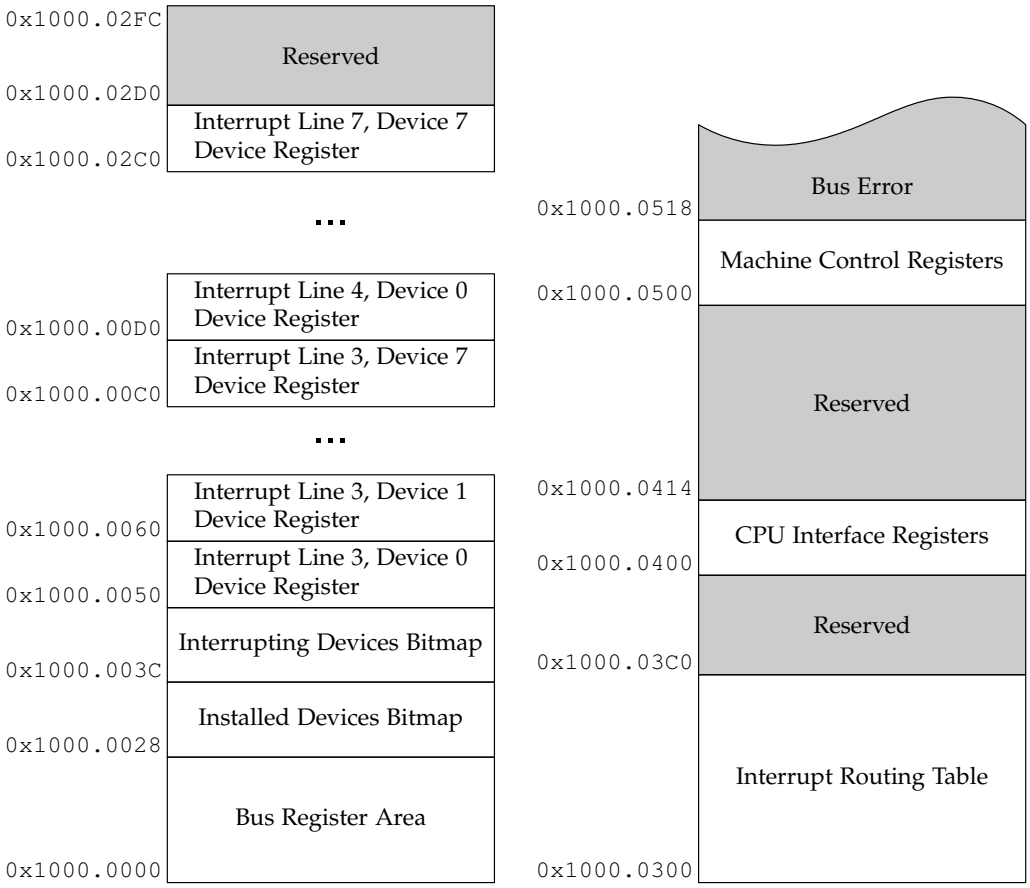


Figure A.8: *Device register memory map.*

Appendix B

Machine Configuration Format

A complete description of the μMPS2 machine configuration format is given below, in the form of a JSON Schema [32]. It is hoped that the schema is reasonably self-documenting—at least to the extent a document of that type can be. Notes have been added in the few places where further explanation was deemed necessary.

Listing B.1: A JSON Schema for the μMPS2 machine configuration format.

```
{
  "type" : "object",
  "properties" : {
    "num-processors" : {
      "type" : "number",
      "description" : "Number of processors",
      "required" : false,
      "minimum" : 1,
      "maximum" : 16,
      "default" : 1
    },
    "clock-rate" : {
      "type" : "number",
      "description" : "Processor/bus clock rate",
      "required" : false,
      "minimum" : 1,
      "maximum" : 99,
      "default" : 1
    },
    "tlb-size" : {
      "type" : "number",
      "description" : "Translation lookaside buffer size",
      "required" : false,
      "minimum" : 4,
      "maximum" : 64,
      "default" : 16
    },
    "num-ram-frames" : {
```

```
29        "type" : "number",
30        "description" : "Number of RAM frames",
31        "required" : false,
32        "minimum" : 8,
33        "maximum" : 512,
34        "default" : 64
35      },
36      "bootstrap-rom" : {
37        "type" : "string",
38        "description" : "Bootstrap ROM image file name",
39        "required" : false,
40        "default" : default bootstrap ROM module (1)
41      },
42      "execution-rom" : {
43        "type" : "string",
44        "description" : "Execution ROM image file name",
45        "required" : false,
46        "default" : default execution ROM module (2)
47      },
48      "boot" : {
49        "type" : "object",
50        "properties" : {
51          "core-file" : {
52            "type" : "string",
53            "description" : "Core file name",
54            "required" : false,
55            "default" : "kernel.core.umps"
56          },
57          "load-core-file" : {
58            "type" : "boolean",
59            "description" : "Load core file on startup",
60            "required" : false,
61            "default" : true
62          }
63        }
64      },
65      "devices" : {
66        "type" : "object",
67        "description" : "Installed devices",
68        "patternProperties" : {
69          "(disk|tape|printer|terminal)[0-7]" : { (3)
70            "type" : "object",
71            "properties" : {
72              "file" : {
73                "type" : "string",
74                "description" : "Device file",
75                "required" : true
76              },
77              "enabled" : {
```

```
78                "type" : "boolean",
79                "description" : "Enable this device",
80                "required" : true
81              }
82            }
83          },
84          "eth[0-7]" : { ③
85            "type" : "object",
86            "properties" : {
87              "file" : {
88                "type" : "string",
89                "description" : "Device file",
90                "required" : true
91              },
92              "enabled" : {
93                "type" : "boolean",
94                "description" : "Enable this device",
95                "required" : true
96              },
97              "address" : {
98                "type" : "string",
99                "description" : "MAC address",
100               "required" : false,
101               "pattern" :
102                 "[A-Fa-f0-9][02468aAcCeE](:([A-Fa-f0-9]{2})){5}"
103             }
104           }
105         }
106       }
107     }
108   }
109 }
```

① The default bootstrap ROM image is `coreboot.rom.umps`, included in the distribution and installed under a configuration-dependent location (typically `/usr/share/umps2` or `/usr/local/share/umps2`).

② The default bootstrap ROM image is `exec.rom.umps`, distributed with μMPS2 and installed under a configuration-dependent location.

③ For example, the object corresponding to the first terminal device is indexed by the key `terminal0` and the one corresponding to the second network device is indexed by `eth1`. Note that all mappings are optional.

Bibliography

[1] M. Goldweber, R. Davoli, and M. Morsiani, "The Kaya OS project and the μMPS hardware emulator," *SIGCSE Bull.*, vol. 37, pp. 49–53, June 2005.

[2] M. Goldweber, R. Davoli, and T. Jonjic, "Supporting operating systems projects using the μMPS2 hardware simulator," in *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, ITiCSE 2012, (New York, NY, USA), pp. 63–68, ACM, 2012.

[3] R. H. Austing, B. H. Barnes, D. T. Bonnette, G. L. Engel, and G. Stokes, "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science," *Commun. ACM*, vol. 22, pp. 147–166, March 1979.

[4] L. Cassel, A. Clements, G. Davies, M. Guzdial, R. McCauley, A. McGettrick, R. Sloan, L. Snyder, P. Tymann, and B. Weide, "Computer science curriculum 2008: An interim revision of cs 2001," 2008.

[5] J. Lions, *Lions' commentary on UNIX 6th edition with source code*. San Jose, CA, USA: Peer-to-Peer Communications, Inc., 1996.

[6] M. J. Bach, *The Design of the UNIX Operating System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[7] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005.

[8] J. Parker, R. Cupper, C. Kelemen, D. Molnar, and G. Scragg, "Laboratories in the Computer Science Curriculum," *Computer Science Education*, vol. 1, no. 3, pp. 205–221, 1990.

[9] N. Titterton and M. Clancy, "Adding some lab time is good, adding more must be better: the benefits and barriers to lab-centric courses," in *proceedings of the 2007 International Conference on Frontiers in Education: Computer Science and Computer Engineering, Las Vegas, NV*, 2007.

[10] M. Morsiani and R. Davoli, "Learning operating systems structure and implementation through the MPS computer system simulator," in *ACM SIGCSE Bulletin*, vol. 31, pp. 63–67, ACM, 1999.

[11] M. Goldweber and R. Davoli, *uMPS2 Principles of Operation*. Lulu Books, 2011.

[12] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 4th ed., 2008.

[13] K. Vollmar and P. Sanderson, "MARS: an education-oriented MIPS assembly language simulator," in *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, SIGCSE '06, (New York, NY, USA), pp. 239–243, ACM, 2006.

[14] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 41–41, USENIX Association, 2005.

[15] J. L. Hennessy, N. Jouppi, F. Baskett, and J. Gill, "MIPS: A VLSI Processor Architecture," tech. rep., Stanford, CA, USA, 1981.

[16] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[17] G. Kane, *MIPS RISC Architecture*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[18] D. Crockford, "The application/json Media Type for Javascript Object Notation (JSON)," *http://www.ietf.org/rfc/rfc4627*, 2006.

[19] "The Red Hat Newlib C Library."
`http://sourceware.org/newlib/libc.html`.

[20] T. Jonjic, "μMPS2 Cross Toolchain Guide."
`http://mps.sourceforge.net/pdf/umps-cross-toolchain-guide.pdf`.

[21] Z. Wan and P. Hudak, "Functional reactive programming from first principles," *SIGPLAN Not.*, vol. 35, pp. 242–252, May 2000.

[22] "Libsigc++ Callback Framework for C++."
`http://libsigc.sourceforge.net/`.

[23] B. Stroustrup, *The C++ Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 1997.

[24] C. Liu, *Smalltalk, Objects, and Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[25] M. Summerfield, *Advanced Qt Programming*. Upper Saddle River, NJ, USA: Prentice Hall Press, 1st ed., 2010.

[26] S. Goderis, *On the Separation of User Interface Concerns: A Programmer's Perspective on the Modularisation of User Interface Code*. PhD thesis, Vrije Universiteit Brussels, 2007-2008.

[27] D. F. Sutherland, A. Greenhouse, and W. L. Scherlis, "The code of many colors: relating threads to code and shared state," *SIGSOFT Softw. Eng. Notes*, vol. 28, pp. 77–83, Nov. 2002.

[28] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification 1.2," 1995.

[29] J. Levine, *Linkers and Loaders*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[30] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 3, no. 4, pp. 34–43, 1995.

[31] J. Hennessy, D. Patterson, and D. Goldberg, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th ed., 2006.

[32] K. Zyp *et al.*, "A JSON media type for describing the structure and meaning of JSON documents," *http://tools.ietf.org/html/draft-zyp-json-schema-03*, 2010.

Acknowledgments

I am grateful to my advisor, Prof. Renzo Davoli, to Mauro Morsiani, and to Prof. Michael Goldweber, for having provided both helpful advice and invaluable feedback during my work on this thesis.